# I/O-Efficient Algorithms for Shortest Path Related Problems

By

Norbert Ralf Zeh

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

April 28, 2002

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

**I/O-Efficient Algorithms for Shortest Path Related Problems**

submitted by

Norbert Ralf Zeh

---

Dr. Frank Dehne
(Director, School of Computer Science)

---

Dr. Anil Maheshwari
(Thesis Co-Supervisor)

---

Dr. Jörg-Rüdiger Sack
(Thesis Co-Supervisor)

---

Dr. Roberto Tamassia
(External Examiner)

Carleton University

April 28, 2002

To My Grandparents

Joseph and Gertrud Leschinsky

*Wish you were here ...*

# Abstract

In this thesis, we study I/O-efficient algorithms for problems related to computing shortest paths in outerplanar and planar graphs and in spanner graphs for point sets in $d$-dimensional space and sets of obstacles in the plane.

In particular, we show in the first part of the thesis that the following problems can be solved in sorting complexity or even in a linear number of I/Os: outerplanarity testing, outerplanar embedding, planarity testing, planar embedding, computing optimal $\varepsilon$-separators of planar and outerplanar graphs, breadth-first search, depth-first search, and single source shortest paths on planar and outerplanar graphs.

In the second part of the thesis, we show that the well-separated pair decomposition of [37] can be computed in sorting complexity. We use this decomposition to construct two types of Euclidean spanners of linear size for point sets in $d$ dimensions. The first spanner is derived in a natural manner from the well-separated pairs in the decomposition. The second spanner is a supergraph of the first spanner. The particular structure of this spanner makes it possible to construct a data structure which can be used to report spanner paths in an I/O-efficient manner. For sets of polygonal obstacles in the plane, we use a subdivision derived from the fair split tree of a point set to compute a planar Steiner spanner of the set of obstacles. Given the results from the first part of the thesis and results from [2, 100, 177], the planarity of the graph can be exploited to compute spanner paths I/O-efficiently and to preprocess the graph so that shortest path queries can be answered and paths in the graph can be traversed in an I/O-efficient manner.

As part of the results in Part I of the thesis, we present improved and much simplified algorithms for computing maximal matchings and maximal independent sets of general undirected graphs. As an additional application of the well-separated pair decomposition, which is at the core of the algorithms presented in Part II, we obtain nearly I/O-optimal algorithms for solving the $K$-nearest neighbor and $K$-closest pair problems for point sets in $d$ dimensions.

# Acknowledgements

First of all, I would like to thank my two supervisors, Anil Maheshwari and Jörg-Rüdiger Sack, for their unfailing support without which I could not have completed the work presented here.

Anil has become more of a friend than a supervisor in these over three years we worked together. His ability to ask the right questions and then listen quietly gave me room to develop my own ideas, guided by his experience expressed in those questions. His broad interest made it virtually impossible to find a research question he would not be interested in. His quiet, yet joyful, manner always made it a pleasure to work with him. He taught me that the best way to get out of a dead end is to laugh about it and then move on.

My co-authors Lars Arge, Tamás Lukovszki, Anil Maheshwari, Ulrich Meyer, Michiel Smid, and Laura Toma have been excellent fellow researchers to work with.

The members of my thesis examination committee deserve my thanks for carefully reading this thesis and for their helpful and encouraging comments. I would also like to thank them for making the defense an entirely pleasant and stress-free experience. Roberto Tamassia deserves special thanks for bringing the previous results on dynamic planarity testing discussed at the end of Chapter 9 to my attention.

I would like to thank Anil Maheshwari, Jörg-Rüdiger Sack, Carleton University, NSERC, and NCE Geoide for their financial support without which I could not have afforded to study at Carleton University.

Though not directly contributing to my studies, there are a few people who helped me in their own way to finish this thesis. Jason Morrison was my brother-in-arms.

We helped each other to deal with our frustrations and shared each other's excitement about a new idea or an accepted publication. Jit Bose always helped when help was needed, in his function as Graduate Advisor of the School of Computer Science and as a friend.

My parents deserve my utmost gratitude for their loving care, for always believing in my ability to finish this work, and for never reproaching me because my work so often made me neglect family matters.

Last but not least, my fiancée Nelly Matsumoto has been with me more than half of the time that I worked on this thesis. She has coped with my grumpiness when immersed in my work. And she has done everything in her power to make my home an excellent working environment. Most importantly though, her boundless love and her unbelievable happiness, which she manages to transmit to everybody she meets, gave me new energy every day.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

Strategies for systematically exploring graphs such as breadth-first search and depth-first search are fundamental for the design and analysis of graph algorithms. Depth-first search, for example, is applied in algorithms for solving such fundamental problems as computing the connected, biconnected, and triconnected components of a given graph [98] and deciding whether a given graph is planar [97]. Shortest path problems arise naturally in areas such as robotics, computational graph theory, and computational geometry. Recent applications include the area of web modelling [32], where depth-first search, breadth-first search, shortest paths, and connected components are used to explore the structure of the web, and Geographic Information Systems (GIS), where many fundamental problems can be solved using graph algorithms. Our interest in shortest path problems arose in the context of computing approximate shortest paths on triangular irregular networks [119, 120, 121], which are a commonly used data structure to represent elevation models in GIS [115, 116].

Web modelling applications and GIS often have to handle massive data sets that do not fit into the main memory of state-of-the-art computers. Recent web crawls, for example, produce graphs of on the order of 200 million vertices and 2 billion edges [32]. Thus, most of the data is stored on disk, while only a small fraction of the data can

reside in main memory at any point in time. In this situation the transfer of data between internal and external memory, and not the internal memory computation, is often the bottleneck of the algorithm, as the seek time of state-of-the-art hard-drives is about six orders of magnitude larger than the time it takes to access a memory location in main memory [147, 171].

The largest portion of the time it takes to transfer a data item between disk and main memory is spent on positioning the read-write head of the disk. Once the head is positioned, successive data items can be read at a reasonable speed [147, 171]. Thus, it is desirable to transfer more than one data element per I/O-operation between disk and main memory, in order to amortize the large seek time over a larger number of transferred data items. This has lead computer architects and operating system designers to partitioning hard disks into blocks of consecutive data items. One block of data can be transferred between main memory and disk in a single I/O-operation. In order to take advantage of the resulting higher throughput, an algorithm should use as many elements read in an I/O-operation as possible in subsequent computation steps. The gain achieved by grouping the data items into blocks is lost if the algorithm uses only few data items per block it reads and discards the remaining items in the block. Thus, in order to be I/O-efficient, an algorithm should organize its computation so that it follows the blockwise manner in which the data is stored in external memory. For most non-trivial problems the design of such algorithms is a challenging task.

Given that an I/O-efficient algorithm should use all data in a block transferred to main memory before loading the next block into main memory, locality of access in such algorithms is desirable, while random access is what is to be avoided. In geometric computations, locality of access can often be ensured by sorting the geometric objects by their positions along one of the coordinate axes and then processing them in this order. For graph algorithms, on the other hand, it is difficult to devise a general scheme to make an algorithm access the vertices of a graph in a blockwise fashion. This is true because a vertex can be connected to any other vertex in the graph. That

is, interactions between vertices that need to be explored by the algorithm are often of a non-local nature.

As a result, earlier algorithms for solving graph problems I/O-efficiently spend $\mathcal{O}(1)$ I/Os per vertex, while the use of I/O-efficient data structures such as queues, stacks, search trees, and priority queues allows these algorithms to access the edges of the graph in a blockwise fashion. For dense graphs, the number of I/Os spent on randomly accessing the vertices of the graph can be amortized over the larger number of edges in the graph, so that these algorithms are I/O-efficient and often optimal for dense graphs. For sparse graphs, however, such an amortization argument cannot be applied, so that the design of I/O-efficient algorithms for sparse graphs is a field with a large number of challenging open problems.

The lack of success in designing algorithms that are I/O-efficient on sparse graphs in general suggests that one should try to exploit the structure of special classes of sparse graphs to design I/O-efficient algorithms for these graph classes. This idea is by no means new, as it has for instance been applied to obtain more efficient internal memory algorithms for shortest path problems on planar graphs [73, 74, 111] or linear time algorithms for problems on graphs of bounded treewidth which are NP-hard in general [17, 27].

The first part of this thesis focuses on this idea. We propose I/O-efficient algorithms for breadth-first search, depth-first search, shortest paths, and related problems on outerplanar and planar graphs. The choice of the problems is motivated by the fact that they are fundamental. They are used as primitives in a wide range of more complex graph algorithms. The choice of the graph classes is motivated by the fact that they are well-studied. They exhibit sufficient structure to be exploited by algorithms designed particularly for these graph classes and are still general enough to hope that graphs arising in real applications may belong to these classes. For example, the graphs studied in the work of [119, 120, 121] are close to being planar, and algorithms for planar graphs can easily be adapted to handle these graphs.

The second part of this thesis focuses on shortest path problems of a geometric nature: the construction of sparse geometric spanner graphs and the reporting of short spanner paths between two query points in these graphs. In internal memory, geometric spanners have been applied successfully to a number of proximity and shortest path problems including the computation of approximate shortest paths among a set of polygonal obstacles in the plane. The success of these graphs in internal memory is based on the fact that they are sparse. Given a spanner which approximates the complete Euclidean graph of a point set or the visibility graph of a set of polygonal obstacles sufficiently well, a spanner path provides a good approximation of the geometric shortest path between its two endpoints. If the spanner is sparse, a shortest path in the spanner can be computed efficiently using Dijkstra's algorithm.

In external memory, the situation is more complicated. While a reduction of the number of edges in the graph certainly leads to a considerable speed-up of shortest path algorithms, we would like to exploit the structure of the spanner graphs, in order to either obtain algorithms that can compute spanner paths in the constructed spanners in $o(1)$ I/Os per vertex, or preprocess the spanner graph so that spanner paths can be reported I/O-efficiently. These are the problems we address in the second part of the thesis.

## 1.2 Summary of the Thesis

In this section, we summarize the results obtained in this thesis. The thesis is divided into two parts. The first part of the thesis is dedicated to problems related to solving breadth-first search (BFS), depth-first search (DFS), and the single source shortest path (SSSP) problem in planar and outerplanar graphs. In the second part, we present algorithms for computing sparse spanners of the complete Euclidean graph of a point set in $d$-dimensional space and the visibility graph of a set of polygonal obstacles in the plane. We also show how to build data structures for these spanners so that approximate shortest path queries between two query vertices can be reported efficiently. Next we discuss the results of each chapter in detail.

| Problem | Previous Result | Our Result |
|---|---|---|
| *General graphs* | | |
| Maximal matching | $\mathcal{O}\left(\frac{\lvert E\rvert}{\lvert V\rvert}\mathrm{sort}(\lvert V\rvert)\log_2\frac{\lvert V\rvert}{M}\right)$ [1] | $\mathcal{O}(\mathrm{sort}(\lvert V\rvert+\lvert E\rvert))$ [128] |
| Maximal independent set | $\mathcal{O}(\lvert V\rvert+\lvert E\rvert)$ | $\mathcal{O}(\mathrm{sort}(\lvert V\rvert+\lvert E\rvert))$ |
| Coloring graphs of bounded degree | $\mathcal{O}(\lvert V\rvert+\lvert E\rvert)$ | $\mathcal{O}(\mathrm{sort}(\lvert V\rvert+\lvert E\rvert))$ [126] |
| *Outerplanar graphs* | | |
| Outerplanarity testing, outerplanar embedding | $\mathcal{O}(N)$ [135] | $\mathcal{O}(\mathrm{perm}(N))$ [127] |
| BFS, DFS, SSSP, weighted $\varepsilon$-separator | $\mathcal{O}(\mathrm{sort}(N)\log N)$ [28, 43] | $\mathcal{O}(\mathrm{scan}(N))$ [127] |
| *Planar graphs* | | |
| Planarity testing | $\mathcal{O}(\mathrm{sort}(N)\log^2 N)$ [112, 43] | $\mathcal{O}(\mathrm{perm}(N))$ [129] |
| Planar embedding | $\mathcal{O}(\mathrm{sort}(N)\log N)$ [145, 43] | $\mathcal{O}(\mathrm{perm}(N))$ [129] |
| BFS, SSSP | $\mathcal{O}(N)$ [111] | $\mathcal{O}(\mathrm{perm}(N))$ [12, 129] |
| DFS | $\mathcal{O}(\mathrm{sort}(N)\log N)$ [105, 43] | $\mathcal{O}(\mathrm{perm}(N))$ [13, 129] |
| Weighted $\varepsilon$-separator | $\mathcal{O}(\mathrm{sort}(N^{1+\varepsilon})\log^3 N)$ [83, 43] | $\mathcal{O}(\mathrm{perm}(N))$ [129] |

**Table 1.1**
Graph algorithms.

## Part I: Graph Algorithms

Our work on graph algorithms focuses mainly on problems on planar and outerplanar graphs which are related to solving BFS, DFS, and the single source shortest path problem on these graphs. However, we also propose a simple framework to derive I/O-efficient solutions for a number of fundamental graph problems on general graphs from the internal memory algorithms for these problems. The solutions for the latter problems are used as primitives in our algorithms for planar graphs as well as in our spanner algorithms discussed in Part II. Table 1.1 summarizes our results. Next, we discuss them in detail.

**Greedy algorithms.** In Chapter 6, we study the properties a greedy graph algorithm has to have, in order to be I/O-efficient. In particular, we want to simulate the algorithm in $\mathcal{O}(\text{sort}(N))$ I/Os using the time-forward processing technique. While it is obvious that the simulation succeeds under the conditions we state, we still obtain interesting new results using this approach. In particular, we obtain simple algorithms for computing maximal independent sets and maximal matchings of arbitrary graphs that outperform existing algorithms for these problems. We also obtain an $\mathcal{O}(\text{sort}(N))$ I/O algorithm for $(\Delta + 1)$-coloring a graph of degree $\Delta$. This work has been published in [126, 128].

**Outerplanar graphs.** In Chapter 7, we study the I/O-complexity of solving BFS, DFS, and SSSP on outerplanar graphs, and of computing small $\varepsilon$-separators of these graphs. We show that these problems can be solved in $\mathcal{O}(\text{scan}(N))$ I/Os, once an outerplanar embedding is given, represented in an appropriate manner. We also show that an outerplanar embedding can be obtained in $\mathcal{O}(\text{perm}(N))$ I/Os and how to augment the embedding algorithm to test any given graph for outerplanarity in the same number of I/Os. We prove $\Omega(\text{perm}(N))$ I/O lower bounds for all of these problems, except outerplanarity testing and computing $\varepsilon$-separators. Previous, more complicated, versions of these results have been published in [127].

**Planar graphs.** In Chapters 8 through 11, we study the I/O-complexity of BFS, DFS, and SSSP on planar graphs, and of computing small $\varepsilon$-separators of these graphs. In Chapter 8, we show that an unweighted $\varepsilon$-separator of a planar graph can be obtained in $\mathcal{O}(\text{sort}(N))$ I/Os, provided that the available amount of main memory is sufficiently large. In Chapter 9, we use this separator algorithm to develop an $\mathcal{O}(\text{perm}(N))$ I/O algorithm to test whether a given graph is planar and if so, compute a planar embedding of the graph. We also show that $\Omega(\text{perm}(N))$ I/Os are required to compute a planar embedding of a planar graph. In Chapter 10, we discuss how to combine these two results with existing results of [9, 12] to solve SSSP and BFS on planar graphs in $\mathcal{O}(\text{sort}(N))$ I/Os and to compute $\varepsilon$-separators of low costs and

small $\varepsilon$-edge separators of these graphs. In Chapter 11, we use the fact that a planar embedding and a BFS-tree of a planar graph can be obtained in $\mathcal{O}(\text{sort}(N))$ I/Os, in order to develop an $\mathcal{O}(\text{sort}(N))$ I/O algorithm for DFS in planar graphs.

The results in Chapters 8 and 10 have been published in [129]. The result in Chapter 9 has been published in [129]. The result in Chapter 11 has been published in [13].

## Part II: Geometric Spanners and Proximity Problems

The first part of the thesis deals with shortest path problems in abstract graphs whose combinatorial structure stems from their close relation to geometric structures. In Part II of the thesis, we turn to the problems of solving proximity problems and answering approximate shortest path queries in geometric domains. In particular, we show that a number of well-known geometric spanner graphs can be constructed I/O-efficiently. We propose techniques to report spanner paths in these graphs in an I/O-efficient manner. The core of these algorithms is an I/O-efficient procedure to construct a well-separated pair decomposition of a point set in $d$ dimensions. We use this decomposition to solve the $K$-nearest neighbor and $K$-closest pair problems I/O-efficiently. Tables 1.2 and 1.3 summarize our results. Next we discuss these results in detail.

**The well-separated pair decomposition and applications.** The well-separated pair decomposition (WSPD) [37] is a powerful tool to solve proximity problems in higher dimensions. In particular, Callahan and Kosaraju [34, 35, 37, 38, 40] show that the WSPD can be used to solve the $K$-nearest neighbor and $K$-closest pair problems in $\mathcal{O}(N \log N + KN)$ and $\mathcal{O}(N \log N + K)$ time, respectively. In Chapter 12, we present $\mathcal{O}(\text{sort}(KN))$ and $\mathcal{O}(\text{sort}(N + K))$ I/O algorithms for these two problems, also using the WSPD.

Another important application of the WSPD is the construction of sparse spanner graphs. In particular, it has been shown in [35] that a $t$-spanner of linear size for a point set in higher dimensions can be derived from a WSPD of the point set.

| Spanner | Previous Result | Our Result |
|---|---|---|
| WSPD-spanner in $\mathbb{R}^d$ | $\mathcal{O}(\text{sort}(N)\log N)$ construction, $\mathcal{O}(\log N)$ path reporting [34, 43] | $\mathcal{O}(\text{sort}(N))$ construction, $\mathcal{O}(\log N)$ path reporting [91] |
| Dumbbell spanner in $\mathbb{R}^d$ | $\mathcal{O}(N\log N)$ construction, $\mathcal{O}(\log N)$ path reporting [18] | $\mathcal{O}(\text{sort}(N))$ construction, $\mathcal{O}(\log N/(DB))$ path reporting [126] |
| Planar Steiner spanner, point sets | $\mathcal{O}(N\log N)$ construction [16] | $\mathcal{O}(\text{sort}(N))$ construction [126] |
| Planar Steiner spanner, obstacles | $\mathcal{O}(N\log N)$ construction [16] | $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ construction [126] |

**Table 1.2**
Algorithms to construct geometric spanners and report spanner paths in these graphs.

In [34, 35, 37, 38, 40], sequential and parallel algorithms for computing a WSPD have been proposed. The sequential algorithm is based on a binary divide-and-conquer approach, which can only be made to run in $\mathcal{O}(\text{scan}(N)\log N)$ I/Os. Simulating the parallel algorithm using the PRAM-simulation technique of [43] leads to an $\mathcal{O}(\text{sort}(N)\log N)$ I/O algorithm. We show in Chapter 12 that an $\mathcal{O}(\text{sort}(N))$ I/O algorithm can be obtained by combining parts of both algorithms with existing paradigms for I/O-efficient algorithms. This combination is non-trivial, and the paradigms for I/O-efficient algorithms are applied in novel, non-standard ways. This work has been published in [91].

**The dumbbell spanner.** A disadvantage of the WSPD-spanner constructed by the algorithm in Chapter 12 is that we do not know how to report spanner paths between query points I/O-efficiently. In Chapter 13, we propose an $\mathcal{O}(\text{sort}(N))$ I/O procedure to construct the dumbbell spanner of [18], which is a supergraph of the WSPD-spanner. This spanner has the desirable property that it can be decomposed into a constant number of trees so that for any two points, there is a spanner path which is

| Problem | Previous Result | | Our Result | |
|---|---|---|---|---|
| $K$-closest pairs | $\mathcal{O}(\text{sort}(N+K)\log N)$ | [34, 39, 43] | $\mathcal{O}(\text{sort}(N+K))$ | [91] |
| $K$-nearest neighbors | $\mathcal{O}(\text{sort}(KN)\log N)$ | [34, 39, 43] | $\mathcal{O}(\text{sort}(KN))$ | [91] |

**Table 1.3**
Algorithms for fundamental proximity problems using the well-separated pair decomposition.

a subgraph of one of these trees. Thus, we can use existing techniques for reporting paths in trees [100, 177] to report spanner paths in this graph. Our algorithm for constructing the dumbbell spanner is based on the algorithm of [18]. Our contribution is to show that the different phases of the algorithm of [18] can be performed I/O-efficiently. This work has been published in [126].

**A planar Steiner spanner.** Even though the dumbbell spanner can be preprocessed so that spanner paths can be reported I/O-efficiently, it is of little use when trying to solve the approximate shortest path problem among polygonal obstacles. Its hierarchical nature makes it difficult to construct this spanner for sets of polygonal obstacles. In [125], it is shown that another $t$-spanner, namely the $\theta$-graph of a set of polygonal obstacles can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os. Unfortunately, it is not known whether there exists a data structure that allows spanner paths in the $\theta$-graph to be reported in an I/O-efficient manner.

The planar Steiner spanner of [16] tries to show a way out of this dilemma. It can be constructed for sets of polygonal obstacles and has the desirable property of being planar. Planarity is a useful property, as planar graphs allow the single source shortest path problem to be solved I/O-efficiently. Also, planar graphs can be preprocessed to answer shortest path queries and traverse paths in these graphs I/O-efficiently [2, 100, 177]. On the other hand, it is known that it is in general impossible to construct a planar graph with spanning ratio less than $\sqrt{2}$ for a given point set. Hence, the construction in [16] reverts to adding additional (Steiner) points to the point set, in order to achieve planarity. They show that a linear number of

such Steiner points is sufficient, so that the complexity of reporting spanner paths does not increase by more than a constant factor.

In Chapter 14, we show how to construct the spanner of [16] for point sets and sets of polygonal obstacles in the plane in an I/O-efficient manner. The construction for point sets is straightforward, once the underlying planar subdivision has been obtained. We argue that a fair split tree, which is the underlying structure of the WSPD, can be used to obtain a planar subdivision which is essentially as good as the subdivision used in [16]. Thus, the spanner for point sets can be constructed I/O-efficiently. For sets of polygonal obstacles, our algorithm simulates the plane sweep of [16]. However, the algorithm of [16] uses two interacting binary search trees to maintain the sweep-line status. In particular, queries on one tree need to be answered immediately to drive the updates of the other tree. This creates problems because the buffer tree [11], which is the only known search tree which achieves optimal I/O-performance in an offline setting, does not support immediate query responses. We show that the sweep-line status can be maintained in a single buffer tree, thereby allowing the plane-sweep to be performed I/O-efficiently. We also believe that the internal memory algorithm obtained by replacing the buffer tree by a standard $(a, b)$-tree is simpler than the original algorithm of [16]. This work has been published in [126].

# Chapter 2

# Model of Computation

Since the analysis of the I/O-complexity of algorithms is not as well-established as the analysis of their running time or their space requirements, we dedicate a few pages to the discussion of the model of computation we adopt in this thesis. We investigate its relationship to well-established models of computation, which allows a comparison between existing algorithms for the problems we consider and our algorithms. We also mention other models that have been proposed for analyzing the I/O-complexity of algorithms, and justify our choice of a model. At the end of the chapter, we present a simple technique to obtain an I/O-efficient algorithm from two I/O-efficient algorithms that solve the same problem. The constructed algorithm achieves the same performance as the faster of the two algorithms on the given input data.

## 2.1 The Parallel Disk Model

The first widely accepted model for analyzing the I/O-complexity of algorithms was the *I/O-model* of Aggarwal and Vitter [6]. In this model, a single processor is equipped with a random access (internal) memory capable of holding $M$ data items and a disk (external memory) of unlimited size. The disk is partitioned into blocks of $B$ consecutive data items. The processor is allowed to perform its computation only on data items held in internal memory. In order to access other data items, the

processor has to make room for these data items by transferring data from internal to external memory and then loading the desired data items into internal memory. This transfer of data is achieved by means of I/O-operations. In a single I/O-operation, the processor can load up to $D$ blocks of data into internal memory, or write up to $D$ blocks of data to disk, where $D \geq 1$ is the number of independent read-write heads that can be used. The complexity of an algorithm in the I/O-model is the number of I/O-operations it performs.

Note that the I/O-model completely ignores the time it takes to perform the actual computation. This is motivated by the fact that an I/O-operation is by about six orders of magnitude slower than a computation step [147, 171]. Thus, an algorithm that performs considerably less I/Os, even at the expense of performing more computation steps, can be expected to be faster than an algorithm performing more I/O-operations, as long as the amount of computation performed by the algorithm stays within reasonable bounds.

The partitioning of the disk into blocks of size $B$ is motivated by the fact that the major share of time spent on an I/O-operation is spent on moving the read-write head to the location of the block. Once the read-write head is at the right location, it takes almost as much time to read $B$ consecutive data items, as it takes to read a single data item. Thus, existing file systems partition the disk into blocks of a certain size, in order to amortize the seek time over a larger number of data items that are read or written.

A characteristic of real disk systems which is not captured by the I/O-model is that on a disk with multiple (independent) read-write heads, there is often only one head per platter. Hence, every head is restricted to accessing the data stored on its platter rather than being able to access any data item. Vitter and Shriver propose an extension of the I/O-model, the *Parallel Disk Model* (PDM) [172], as a more realistic model to describe existing disk systems which takes the restriction just described into account. This model is now the most widely accepted model for the design and analysis of I/O-efficient algorithms. It is also the model we adopt in this thesis. In

this model, $D \geq 1$ independent disks are attached to the processor. Each disk is assumed to have a single read-write head. This is not a restriction, since the different platters of disks with independent read-write heads can be modelled as separate disks. As in the I/O-model, an I/O-operation can transfer up to $D$ blocks of data between internal and external memory; but now this is allowed only if each of these blocks is read from or written to a different disk.

A number of simulation techniques have been proposed which allow algorithm designers to benefit from the higher practicality of the PDM and the simplicity of the I/O-model at the same time. Most notably, Sanders et al. [152] propose a randomized technique to achieve optimality for algorithms designed in the I/O-model when run on a machine with multiple disks. A simple, suboptimal, deterministic technique is *disk-striping* [149]. Using this technique, the $D$ disks are viewed as one large "virtual disk" of block size $DB$, where the $i$-th block of the "virtual disk" contains the $i$-th block of each of the $D$ disks.

## 2.2 Relation to Standard Models

### 2.2.1 RAM Algorithms

**The model.** The most extensively studied and most broadly used model for designing algorithms and analyzing their performance is the *random access machine* (RAM). Algorithms designed for this model consist of a single thread of execution. That is, the instructions of the algorithm are executed one at a time. Legal instructions are elementary arithmetic and logical operations, instructions to read and write data from or to memory, and elementary control constructs to realize branching, loops, and recursion. Each of these operations is assumed to take $\mathcal{O}(1)$ time on a data item representable by $\mathcal{O}(\log N)$ bits, where $N$ is the size of the input. Thus, in order to estimate the time it takes an algorithm to solve a given problem, it is sufficient to count the number of operations executed by the algorithm.

A number of variations of this model have been proposed, which can be distinguished based on the set of arithmetic operations that are considered primitives of the machine. Since most operations of a more powerful RAM model can be simulated at a small, though non-constant, cost in a weaker RAM model, these variations are of limited relevance in the context of this thesis, as we ignore the computation cost of our algorithms. Nevertheless, the operations performed by our algorithms fit in the *algebraic model* of computation, which allows only multiplication, division, addition, and subtraction as primitive arithmetic operations of the machine. In particular, the floor function is not considered a primitive.

Our geometric algorithms assume that the machine words that can be manipulated in $\mathcal{O}(1)$ time per operation are real numbers. This avoids the hassle of dealing with precision problems. However, these issues would have to be addressed when implementing our algorithms.

**The I/O-complexity of RAM algorithms.** Since every computation step of a RAM algorithm can access at most one data item which is not in internal memory, every computation step of the algorithm takes at most one I/O. Hence, any RAM algorithm which takes $T(N)$ time performs at most $T(N)$ I/Os. This simple observation becomes important in combination with contraction or sampling techniques where preliminary information about the given problem instance is gathered by applying a RAM algorithm to a problem instance of reduced size. Due to the reduced size of the sample or the contracted problem instance, even spending one I/O per computation step is I/O-efficient in terms of the size of the uncontracted or unsampled problem instance.

### 2.2.2 PRAM Algorithms

**The model.** The *parallel random access machine* (PRAM) is a generalization of the RAM which allows programs to consist of several threads of execution. The cost of an operation is the same as in the RAM model. However, the machine now

consists of multiple RAM-type processors with access to a global memory shared by all processors. The processors are employed to collectively solve a given problem. The time it takes to solve the problem is the maximal amount of time spent by any one of the processors. Usually the goal is to design algorithms that take time polylogarithmic in the size of the input using a polynomial number of processors.

**The I/O-complexity of PRAM algorithms.** An interesting approach for designing I/O-efficient algorithms has been proposed by Chiang et al. [43]. This approach derives I/O-efficient algorithms from existing PRAM algorithms. It is based on the observation that $P$ simultaneous computation steps, one per processor, can be simulated in $\mathcal{O}\left(\text{sort}(P) + \frac{N}{DB}\right)$ I/Os[1] as follows: Assume that the contexts of the $P$ processors are stored consecutively on disk, and the content of the shared memory is stored on disk. In order to simulate the next computation step of each of the processors, assume that this computation step consists of a read access to shared memory, followed by an arithmetic or logical operation, which in turn is followed by a write access to shared memory. Instead of accessing the data item for processor number $i$ directly, the list of processor contexts is scanned in $\mathcal{O}\left(\frac{P}{DB}\right)$ I/Os to produce a list of read requests. These read requests are sorted by their memory addresses. In a single scan of the sorted list of read requests and the memory representation, the content of the requested memory location is assigned to each read request. Now the list of read requests is re-sorted by their originating processors. In a single scan of the sorted list of read requests and the list of processor contexts, the requested data is transferred to the contexts of the processors. Now each processor performs its local computation and generates a write request. The list of write requests is processed in a manner similar to the processing of read requests.

As each step of the PRAM algorithm can be realized in $\mathcal{O}\left(\text{sort}(P) + \frac{N}{DB}\right)$ I/Os, a $P$-processor PRAM algorithm which runs in $T(N, P)$ time can be realized in

[1] $\text{sort}(N) = \Theta\left(\frac{N}{DB} \log_{\frac{M}{B}} \frac{N}{B}\right)$ denotes the number of I/Os it takes to sort a list of $N$ elements.

$\mathcal{O}\left(T(N,P)\left(\text{sort}(P)+\frac{N}{DB}\right)\right)$ I/Os. In particular, any $\mathcal{O}(\log_2 N)$ time algorithm using a linear number of processors takes $\mathcal{O}(\text{sort}(N)\log_2 N)$ I/Os when simulated in external memory. Note, however, that even an algorithm using $\mathcal{O}\left(\frac{N}{\log N}\right)$ processors will take $O\left(\frac{N}{DB}\log_2 N\right)$ I/Os. If the computation steps of the algorithm correspond to a recursion such that the amount of data processed in each recursive step as well as the number of processors participating in each recursive step are geometrically decreasing, the algorithm takes only $\mathcal{O}\left(\text{sort}(P)+\frac{N}{DB}\right)$ I/Os. The latter special case of this simulation technique has been employed to derive a number of I/O-optimal graph algorithms for sparse graphs from existing PRAM-algorithms [43].

## 2.3 Related Models of Computation

While the PRAM model tries to model shared memory machines, coarse grained models of parallel computation model message passing systems, where the only means of communication between processors is the exchange of messages between them. Coarse grained models include the BSP model [170], the BSP* model [21], and the CGM model [57]. In [56], the EM-BSP, EM-BSP*, and EM-CGM models have been proposed as extensions of the BSP, BSP*, and CGM models which allow the analysis of the performance of parallel algorithms in terms of computation, communication, and I/O.

Even though I/O-efficient algorithms in terms of the PDM can be obtained from efficient EM-BSP, EM-BSP*, or EM-CGM algorithms by simulating multiple processors on a single processor, the relevance of work in these models to the work presented here is limited for the following reasons: Few algorithms have been developed directly in any of these models. Rather, efficient algorithms for the EM-CGM model are obtained through a general, optimal, simulation of CGM-algorithms in this model. Only few algorithms for more complicated graph or geometric problems have been developed in the BSP or BSP* model, as the large number of parameters used by these models to describe the given machine make them difficult to use for analyzing the complexity of non-trivial algorithms. Algorithms designed for the CGM model are often

obtained by simulating an existing PRAM algorithm for a certain number of communication rounds and then applying a specialized CGM-algorithm to a subproblem of reduced size or complexity. Applying the PRAM-simulation technique described in the previous section directly to the PRAM algorithm, we obtain algorithms of comparable or better I/O-performance without cascading multiple simulation techniques (PRAM $\rightarrow$ CGM $\rightarrow$ EM-CGM $\rightarrow$ PDM).

## 2.4 Dealing with Performance Trade-offs

In this section, we discuss a simple technique for the following problem: Given two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ that solve a problem $\mathcal{P}$, it may be that there are some instances of problem $\mathcal{P}$ where algorithm $\mathcal{A}_1$ performs better and some instances where algorithm $\mathcal{A}_2$ outperforms algorithm $\mathcal{A}_1$. We want to construct an algorithm $\mathcal{A}$ which achieves the performance of the better of the two algorithms on the given input instance without knowing in advance which algorithm will perform better. The following lemma provides the tool to achieve this.

**Lemma 2.1** *Given two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ that solve a problem $\mathcal{P}$ in $\mathcal{I}_1(N)$ and $\mathcal{I}_2(N)$ I/Os using $S_1(N)$ and $S_2(N)$ space, respectively, there exists an algorithm $\mathcal{A}$ that solves problem $\mathcal{P}$ in $\mathcal{O}(\min(\mathcal{I}_1(N), \mathcal{I}_2(N)))$ I/Os and $\mathcal{O}(S_1(N) + S_2(N))$ space, provided that* $\min(\mathcal{I}_1(N), \mathcal{I}_2(N)) = \Omega(\mathrm{scan}(N))$.

*Proof.* Given an instance $P$ of problem $\mathcal{P}$, we create two identical copies $P_1$ and $P_2$ of instance $P$. This takes $\mathcal{O}(\mathrm{scan}(N))$ I/Os. Now we run algorithm $\mathcal{A}_1$ on instance $P_1$, and simultaneously run algorithm $\mathcal{A}_2$ on instance $P_2$. When algorithm $\mathcal{A}_1$ cannot proceed without performing an I/O-operation, we let algorithm $\mathcal{A}_1$ perform this I/O-operation and then switch to algorithm $\mathcal{A}_2$. When algorithm $\mathcal{A}_2$ cannot proceed without performing an I/O-operation, we let algorithm $\mathcal{A}_2$ perform this I/O-operation and then switch back to algorithm $\mathcal{A}_1$. We stop this procedure as soon as one of the two algorithms finishes. The I/O-complexity of this procedure is $\mathcal{O}(\min(\mathcal{I}_1(N), \mathcal{I}_2(N)))$. The required space is $\mathcal{O}(S_1(N) + S_2(N))$. □

# Chapter 3

# Preliminaries

In this chapter, we introduce the common terminology and notation used throughout the thesis. Specific definitions that apply only to particular chapters are provided as needed in each chapter.

## 3.1 I/O-Complexities

We use the following shorthands for the I/O-complexities of sorting, permuting, and scanning a list of $N$ data items:

$$\begin{aligned}\mathrm{sort}(N) &= \Theta\left(\frac{N}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)\\ \mathrm{perm}(N) &= \Theta(\min(\mathrm{sort}(N), N))\\ \mathrm{scan}(N) &= \Theta\left(\frac{N}{DB}\right)\end{aligned}$$

These shorthands have been introduced in the literature because the I/O-complexities of these operations arise frequently in the analysis of algorithms.

Firstly, scanning, sorting, and permuting a list of data items are fundamental operations that arise as subproblems in many algorithms. In particular, algorithms that are designed to be I/O-efficient often sort the data in an appropriate order and then scan the list of data items in their order of appearance, while an equivalent RAM

algorithm would be allowed to access the data items in a random fashion without paying a performance penalty.

Secondly, these I/O-complexities have important relations to well-known time-complexities in the RAM model. In particular, $\mathrm{sort}(N)$ is the equivalent of the $\Theta(N \log N)$ time bound for sorting $N$ data items. Usually, if a problem can be solved in $\mathcal{O}(N \log N)$ time, we hope to be able to design an algorithm which solves this problem in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. The scanning bound is the equivalent of the linear time bound for this problem in the RAM model. Hence, we refer to $\mathrm{scan}(N)$ as a linear number of I/Os, while we consider $N$ I/Os to be superlinear. This point of view is reasonable because an algorithm that spends $N$ I/Os to solve a problem of size $N$ does not utilize the throughput of the I/O-system. Finally, the permutation bound is interesting, as it is superlinear, while permuting $N$ data items in the RAM model takes linear time. This implies that a superlinear lower bound can be shown for the number of I/Os it takes to solve many non-trivial problems that can be solved in linear time in the RAM model, only because these problems contain some permutation problem as a subproblem.

## 3.2 Definitions

In this section, we introduce standard definitions and concepts related to graphs. These concepts are well-established in the literature. Even though different definitions are used for the same concepts in the literature, they are all similar to the ones presented here. Our definitions are closest to the ones used in [69, 93]. The definition of the triconnected components of a graph is taken from [97].

### 3.2.1 Graphs

A *(multi)graph* is an ordered pair $G = (V, E)$ of a set $V$ and a multiset $E$. Graph $G$ is *simple* if $E$ is a set, i.e., every element of $E$ appears exactly once in $E$. The elements of $V$ are the *vertices* of $G$; the elements of $E$ are its *edges*. An edge $e \in E$ is a pair of

vertices $v, w \in V$, $v \neq w$. If the pair is ordered, we write $e = (v, w)$ and call edge $e$ *directed*. Otherwise, we write $e = \{v, w\}$ and call edge $e$ *undirected*. Graph $G$ is directed or undirected if all its edges are directed or undirected, respectively. If $G$ is not simple, it may be necessary to distinguish between multiple edges with the same endpoints. To do this, we assign a unique number $i$ to every edge $\{v, w\}$ or $(v, w)$ and refer to this edge by the triple $(v, w, i)$. If $G$ is undirected, triples $(v, w, i)$ and $(w, v, i)$ refer to the same edge. For an edge $e = (v, w)$ or $e = \{v, w\}$, vertices $v$ and $w$ are the *endpoints* of $e$. If $e$ is directed, $v$ is the *source* and $w$ is the *target* of $e$. Vertices $v$ and $w$ are said to be *adjacent*. Edge $e$ is *incident* to $v$ and $w$.

For a vertex $v$ of an undirected graph $G = (V, E)$, the *neighborhood* of $v$ is the set $\Gamma_G(v) = \{w \in V : \{v, w\} \in E\}$. The *degree* $\deg_G(v)$ of $v$ is defined as the number of edges incident to $v$. For a vertex $v$ of a directed graph $G = (V, E)$, the *in-neighborhood* of $v$ is the set $\Gamma_G^-(v) = \{u \in V : (u, v) \in E\}$. The *out-neighborhood* of $v$ is the set $\Gamma_G^+(v) = \{w \in V : (v, w) \in E\}$. The *neighborhood* of $v$ is the set $\Gamma_G(v) = \Gamma_G^-(v) \cup \Gamma_G^+(v)$. The *in-degree* $\deg_G^-(v)$ of $v$ is the number of edges with target $v$. The *out-degree* $\deg_G^+(v)$ of $v$ is the number of edges with source $v$. The *degree* of $v$ is defined as $\deg_G(v) = \deg_G^-(v) + \deg_G^+(v)$. If the graph $G$ is clear from the context, we write $\Gamma(v), \deg(v), \ldots$ instead of $\Gamma_G(v), \deg_G(v), \ldots$. The *degree* $\deg(G)$ of a graph $G$ is defined as the maximum degree of all its vertices.

A *subgraph* of a graph $G = (V, E)$ is a graph $H = (W, F)$ with $W \subseteq V$ and $F \subseteq E$. For a subset $W \subseteq V$ of vertices, the subgraph *induced* by $W$ is the graph $G[W] = (W, \{\{v, w\} \in E : v, w \in W\})$. Analogously, the subgraph induced by a subset $F \subseteq E$ of edges is the graph $G[F] = (\{v, w : \{v, w\} \in F\}, F)$. For a set $W$ of vertices, we define the graph $G - W$ as the graph $G[V \setminus W]$. For a single vertex $v$, we define $G - v = G - \{v\}$. For a set $F$ of edges, $G - F = G[E \setminus F]$. For a graph $H = (W, F)$, we define $G - H = G - F$. For a graph $G = G_1 \cup G_2$ such that $V(G_1) \cap V(G_2) = W$ and a graph $G_1'$ with $V(G_1') \cap V(G_2) = W$, let $G[G_1/G_1']$ be the graph $G_1' \cup G_2$. Intuitively, $G[G_1/G_1']$ is the graph obtained from $G$ by replacing subgraph $G_1$ with graph $G_1'$. Similar definitions apply for directed graphs.

A *path* is a graph $P = (V, E)$, where $V = \{v_0, \ldots, v_k\}$ and $E = \{\{v_{i-1}, v_i\} : 1 \leq i \leq k\}$. In this case, we write $P = (v_0, \ldots, v_k)$ and call $v_0$ and $v_k$ the *endpoints* of $P$. If $P$ is directed, we call $v_0$ the *source* and $v_k$ the *target* of $P$. For a path $P = (v_0, \ldots, v_k)$ and two indices $0 \leq i \leq j \leq k$, we define $P(v_i, v_j)$ as the subpath $(v_i, \ldots, v_j)$ of $P$. We call path $P$ *simple* if vertices $v_0, \ldots, v_{k-1}$ and vertices $v_1, \ldots, v_k$ are pairwise distinct. Path $P$ is a *cycle* if $v_0 = v_k$. In this case, we write $P = (v_0, \ldots, v_{k-1})$. The *size* of a path is the number of edges in $P$. Directed paths and cycles are defined analogously.

A *tree* $T = (V, E)$ is an undirected graph with $|E| = |V| - 1$ and so that for every pair of vertices $v, w \in V$, there is a path in $T$ with endpoints $v$ and $w$. In particular, tree $T$ does not contain cycles, and the path between any two vertices $v$ and $w$ is unique. A *subtree* of $T$ is a subgraph of $T$ which is itself a tree. A tree $T = (V, E)$ is *rooted* if it has a distinguished *root vertex* $r$. In this case, we call a vertex $v$ an *ancestor* of another vertex $w$, and $w$ a *descendant* of $v$, if $v$ is on the unique path from $r$ to $w$ in $T$. We call $v$ the *parent* of $w$, denoted by $p_T(w)$, and $w$ a *child* of $v$, if $v$ is an ancestor of $w$ and $\{v, w\} \in E$. For two vertices $v$ and $w$, the *lowest common ancestor* $LCA_T(v, w)$ is the vertex $u \in T$ such that $u$ is an ancestor of $v$ and $w$, and no descendant of $u$ has this property. For a vertex $v \in T$, we define $T(v)$ to be the smallest subtree of $T$ which contains all descendants of $v$. For a subtree $T'$ of $T$, we define the *root* of $T'$ as the unique vertex $r' \in T'$ so that $p_T(r') \notin T'$.

A graph $G = (V, E)$ is *bipartite* if $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, and for every edge $\{v, w\} \in E$, $v \in V_1$ and $w \in V_2$. In this case, we write $G = (V_1, V_2, E)$. Observe that every cycle in a bipartite graph is of even size.

We denote the complete graph with $n$ vertices as $K_n = (\{1, \ldots, n\} : \{\{i, j\} : 1 \leq i < j \leq n\})$. We denote the complete bipartite graph with $m + n$ vertices as $K_{m,n} = (\{1, \ldots, m\}, \{m + 1, \ldots, m + n\}, \{\{i, j + m\} : 1 \leq i \leq m \text{ and } 1 \leq j \leq n\})$.

The *contraction* of an edge $e = \{v, w\}$ in a graph $G = (V, E)$ is the operation of replacing vertices $v$ and $w$ by a new vertex $x$ and every edge $\{y, u\} \in E$, $y \in \{v, w\}$ and $u \notin \{v, w\}$, by an edge $\{x, u\}$.

### 3.2.2 Weighted Graphs and Shortest Paths

We call a graph $G = (V, E)$ *weighted* if we are given an assignment $\omega : V \to \mathbb{R}$ or $\omega : E \to \mathbb{R}$ of weights to the vertices or edges of $G$. If necessary, we distinguish between these two possibilities by calling $G$ *vertex-weighted* or *edge-weighted*. For a vertex-weighted graph $G$, the weight of a subgraph $H$ of $G$ is defined as $\omega(H) = \sum_{v \in H} \omega(v)$. Similarly, $\omega(H) = \sum_{e \in H} \omega(e)$ for an edge-weighted graph $G$. For a path $P$ in an edge-weighted graph $G$, we also refer to the weight of $P$ as its *length*. We call a subgraph of $G$ *positive* or *negative* if its weight is positive or negative, respectively.

Given an edge-weighted graph $G = (V, E)$, a *shortest path* from a vertex $v$ to a vertex $w$ is a path of minimum length among all paths with source $v$ and target $w$ in $G$. Such a shortest path is well-defined only if $G$ does not contain a negative cycle which contains a vertex on a path from $v$ to $w$. In this case, we define the distance from $v$ to $w$ in $G$ as $\text{dist}_G(v, w) = \omega(\Pi(v, w))$, where $\Pi(v, w)$ is a shortest path from $v$ to $w$.

### 3.2.3 Connectivity of Graphs

An undirected graph $G = (V, E)$ is *connected* if for every two vertices $v, w \in V$, there exists a path $P \subseteq G$ with endpoints $v$ and $w$. The *connected components* of a graph $G = (V, E)$ are its maximal connected subgraphs.

For a connected graph $G = (V, E)$, a vertex $v \in V$ is a *cutpoint* of $G$ if $G - v$ is disconnected. Graph $G$ is *biconnected* if it does not have any cutpoints. The *biconnected components* or *bicomps* of a connected graph $G$ are the maximal biconnected subgraphs of $G$. For a disconnected graph, we define its bicomps to be the bicomps of its connected components. The bicomps and cutpoints of a connected graph $G$ define a tree $T_{\text{bic}}(G)$, which we call the *bicomp-cutpoint-tree* of $G$. Let $v_1, \ldots, v_k$ be the cutpoints of $G$, and $\mathcal{B}_1, \ldots, \mathcal{B}_q$ the bicomps of $G$. Then the vertex set of $T_{\text{bic}}(G)$ contains vertices $v_1, \ldots, v_k$ as well as $q$ *bicomp vertices* $\beta_1, \ldots, \beta_q$. There is an edge $\{v_i, \beta_j\}$ in $T_{\text{bic}}(G)$ if $v_i \in \mathcal{B}_j$. It is easy to verify that $T_{\text{bic}}(G)$ is indeed a tree. For two vertices

$v \in \mathcal{B}_i$ and $w \in \mathcal{B}_j$, every path from $v$ to $w$ in $G$ contains all cutpoints on the path from $\beta_i$ to $\beta_j$ in $T_{\text{bic}}(G)$. Often, tree $T_{\text{bic}}(G)$ is rooted by choosing a bicomp vertex $\beta_r$ as the root of $T_{\text{bic}}(G)$. In this case, we call $\mathcal{B}_r$ the *root bicomp* of $G$. The *parent cutpoint* of a bicomp $\mathcal{B}_i \neq \mathcal{B}_r$ is the cutpoint $v$ so that $v = p(\beta_i)$. The *parent bicomp* of bicomp $\mathcal{B}_i$ is the bicomp $\mathcal{B}_j$ so that $\beta_j = p(p(\beta_i))$.

An *ear-decomposition* $\mathcal{E} = \langle P_0, \ldots, P_k \rangle$ of a biconnected graph $G$ is a decomposition of $G$ into simple paths $P_0, \ldots, P_k$ such that $\bigcup_{j=0}^{k} P_j = G$, $P_0$ consists of a single edge, the endpoints of every path $P_i$, $i \geq 1$, are in $G_{i-1}$, and no other vertices of $P_i$ are in $G_{i-1}$, where $G_{i-1} = \bigcup_{j=0}^{i-1} P_j$. The paths $P_j$ are called *ears*. An *open ear-decomposition* is an ear-decomposition such that for every ear, the two endpoints are distinct. We call an ear $P_j$, $j > 0$, *trivial* if it consists of a single edge. Otherwise, we call ear $P_j$ *non-trivial*. Ear $P_0$ is always considered non-trivial.

**Lemma 3.1 (Whitney [175])** *A graph $G = (V, E)$ is biconnected if and only if it has an open ear-decomposition.*

Given a graph $G = (V, E)$ and a subgraph $H = (W, F)$ of $G$, the bridges of $H$ are defined as follows: Consider the connected components of $G - V(H)$. Let $K$ be such a component. Then $K$ defines a *non-trivial bridge* of $H$ which is the subgraph of $G$ induced by all edges incident to vertices in $K$. A *trivial bridge* is an edge in $G - H$ with both endpoints in $H$. The trivial and non-trivial bridges are the *bridges* of $H$ in $G$.

A pair $\{v, w\}$ of vertices of a biconnected graph $G$ is a *separation pair* if graph $H = (\{v, w\}, \emptyset)$ has at least two non-trivial bridges in $G$ or at least three bridges, one of which is non-trivial. We refer to the bridges of $H$ as the bridges of pair $\{v, w\}$. If pair $\{v, w\}$ has at least two non-trivial bridges, we call $\{v, w\}$ a *non-trivial separation pair*. If $G$ is a simple graph, then all separation pairs of $G$ are non-trivial. Graph $G$ is *triconnected* if it does not have a separation pair.

Given a separation pair $\{v, w\}$ with bridges $B_1, \ldots, B_q$, the *split* $s(v, w, i)$ chooses two graphs $B'$ and $B''$ such that $B' = B_1 \cup \cdots \cup B_{q'}$, $B'' = B_{q'+1} \cup \cdots \cup B_q$, $|E(B')| \geq 2$ and $|E(B'')| \geq 2$, and creates two graphs $G_1 = (V(B'), E(B') \cup \{(v, w, i)\})$

and $G_2 = (V(B''), E(B'') \cup \{(v, w, i)\})$ from $G$. Edge $(v, w, i)$ is called the *virtual edge* corresponding to split $s(v, w, i)$. The *split components* of $G$ are defined as the graphs obtained by recursively splitting $G_1$ and $G_2$ until there are no more separation pairs. There are three types of split components: (1) triconnected simple graphs, (2) triple bonds (two vertices with three edges between them), and (3) triangles. The split components of a biconnected graph are not necessarily unique.

The *merge* $m(v, w, i)$ of two graphs $G_1$ and $G_2$ sharing a virtual edge $(v, w, i)$ constructs a graph $G = (V(G_1) \cup V(G_2), (E(G_1) \setminus \{(v, w, i)\}) \cup (E(G_2) \setminus \{(v, w, i)\}))$ from $G_1$ and $G_2$. A graph $G$ can be reconstructed from its split components by recursive application of merge operations. To construct the triconnected components of a biconnected graph $G$, partition $G$ into a set of split components, then merge bonds sharing virtual edges until no two bonds share a virtual edge, and merge simple cycles sharing virtual edges until no two simple cycles share a virtual edge. The resulting graphs are the *triconnected components* or *tricomps* of $G$. If $G$ is not biconnected, the tricomps of $G$ are the triconnected components of its bicomps. The triconnected components of $G$ are unique and of three types: (1) triconnected simple graphs, (2) bonds, and (3) simple cycles. The separation pairs corresponding to the remaining virtual edges are the *Tutte pairs* of $G$.

The recursive definition of the tricomps of a biconnected graph $G$ gives rise to a tree $T_{\text{tri}}(G)$, which we call the *tricomp tree* of $G$. If $\mathcal{T}_1, \ldots, \mathcal{T}_q$ are the tricomps of $G$, $T_{\text{tri}}(G)$ contains vertices $\tau_1, \ldots, \tau_q$. There is an edge $\{\tau_i, \tau_j\}$ in $T_{\text{tri}}(G)$ if tricomps $\mathcal{T}_i$ and $\mathcal{T}_j$ share a virtual edge. In particular, there exists a bijection between the edges of $T_{\text{tri}}(G)$ and the Tutte pairs of $G$. Every path between two vertices $v \in \mathcal{T}_i$ and $w \in \mathcal{T}_j$ contains at least one member of each of the Tutte pairs corresponding to the edges on the path from $\tau_i$ to $\tau_j$ in $T_{\text{tri}}(G)$.

### 3.2.4 Special Vertex and Edge Sets

An *independent set* is a subset $I \subseteq V$ of the vertices of graph $G = (V, E)$ so that no two vertices in $I$ are adjacent in $G$. Independent set $I$ is *maximal* if there is no set $I'$,

$I \subset I' \subseteq V$, which is independent. That is, every vertex in $V \setminus I$ is adjacent to at least one vertex in $I$. A *matching* is a subset $\mathcal{M} \subseteq E$ of the edges of graph $G = (V, E)$ so that no two edges in $\mathcal{M}$ share an endpoint. Matching $\mathcal{M}$ is *maximal* if there is no matching $\mathcal{M}'$, $\mathcal{M} \subset \mathcal{M}' \subseteq E$. That is, every edge in $E \setminus \mathcal{M}$ shares an endpoint with at least one edge in $\mathcal{M}$.

### 3.2.5 Forests and Spanning Graphs

A *forest* $F$ is a graph whose connected components are trees. We call $F$ *rooted* if every tree in $F$ is rooted.

A *spanning graph* of a connected graph $G = (V, E)$ is a connected subgraph $H = (V, F)$ of $G$. If $H$ is a tree, we call $H$ a *spanning tree* of $G$. A *spanning forest* of a graph $G$ is a subgraph of $G$ which contains a spanning tree for every connected component of $G$.

A spanning graph $H = (V, F)$ of an edge-weighted graph $G = (V, E)$ is called a *$t$-spanner* of $G$, for some $t > 1$, if for any two vertices $v$ and $w$ in $G$, $\text{dist}_H(v, w) \leq t \cdot \text{dist}_G(v, w)$. We call the parameter $t$ the *spanning ratio* or *stretch factor* of $H$. A path $P$ of length at most $t \cdot \text{dist}_G(v, w)$ from $v$ to $w$ in $H$ is called a *$t$-spanner path*. For every pair of vertices $v, w \in G$, consider the $t$-spanner path in $H$ from $v$ to $w$ with the minimum number of edges. Then the *spanner diameter* of $H$ is the maximal number of edges of these shortest $t$-spanner paths between all pairs of vertices $v, w \in G$.

### 3.2.6 Planarity and Outerplanarity of Graphs

Graphs as defined above are abstract combinatorial objects. They are usually visualized by drawing them in $\mathbb{R}^2$ or $\mathbb{R}^3$, representing every vertex $v$ as a distinct point $\mathcal{E}(v)$ and every edge $e$ with endpoints $v$ and $w$ as a contiguous curve $\mathcal{E}(e)$ with endpoints $\mathcal{E}(v)$ and $\mathcal{E}(w)$. We call a drawing $\mathcal{E}(G)$ of graph $G$ in the plane a *topological planar embedding* of $G$ if for any two edges $e_1 = \{v, w\} \in G$ and $e_2 = \{x, y\} \in G$, $\mathcal{E}(e_1) \cap \mathcal{E}(e_2) = \{\mathcal{E}(v), \mathcal{E}(w)\} \cap \{\mathcal{E}(x), \mathcal{E}(y)\}$. That is, no two edges intersect, except

at their endpoints. We call graph $G$ *planar* if it has a topological planar embedding. Given a topological embedding $\mathcal{E}(G)$ of $G$, the *faces* of $\mathcal{E}(G)$ are the maximal connected regions of $\mathbb{R}^2 - \mathcal{E}(V \cup E)$. For a face $f$, the *boundary* of face $f$ is the set of vertices $v$ and edges $e$ so that $\mathcal{E}(v)$ and $\mathcal{E}(e)$ are contained in the closure of face $f$. A triangulation $\Delta(G)$ of a planar graph $G$ is a planar supergraph of $G$ so that all faces in a planar embedding of $\Delta(G)$ are bounded by three edges. The *dual* $G^*$ of a planar embedding $\hat{G}$ of a planar graph $G$ is defined as follows: $G^*$ contains a vertex $f^*$ for every face $f$ of $\hat{G}$. There is an edge $\{f_1^*, f_2^*\}$ in $G^*$ if faces $f_1$ and $f_2$ share an edge, i.e., there is an edge which is on the boundary of both $f_1$ and $f_2$.

A graph $G = (V, E)$ is *outerplanar* if it has a topological embedding $\mathcal{E}(G)$ so that there exists a face $f$ of $\mathcal{E}(G)$ which has all vertices of $G$ on its boundary. We refer to $f$ as the *outer* face of $G$. Euler's formula states that for every planar graph, $|V| + |F| - |E| = 2$. This implies in particular that $|E| \leq 3|V| - 6$, for every planar graph, and $|E| \leq 2|V| - 3$, for every outerplanar graph.

Combinatorial characterizations of these two classes of graphs are provided by the following two results, where two graphs are *homeomorphic* if they can both be obtained from the same graph using edge splits. An *edge split* is the operation of replacing an edge $\{v, w\}$ by two edges $\{v, x\}$ and $\{x, w\}$, where $x$ is a new vertex.

**Theorem 3.1 (Kuratowski, e.g. [69])** *A graph $G = (V, E)$ is planar if and only if it does not have a subgraph which is homeomorphic to $K_5$ or $K_{3,3}$.*

**Theorem 3.2 (e.g. [93])** *A graph $G = (V, E)$ is outerplanar if and only if it does not have a subgraph which is homeomorphic to $K_4$ or $K_{2,3}$.*

Using Theorems 3.1 and 3.2, it is not hard to show that every graph which can be obtained from a planar or outerplanar graph by means of edge contractions is itself planar or outerplanar, respectively. Note that an edge contraction is *not* the inverse operation of an edge split.

Most algorithms for planar and outerplanar graphs do not require a topological embedding of these graphs. They only use the order of the edges incident to every

vertex clockwise around that vertex which is induced by the topological embedding. A *combinatorial embedding* $\hat{G}$ of a planar or outerplanar graph $G$ is a representation of the orders of edges around all vertices of $G$ which are induced by a topological embedding of $G$. It is easy to see that for two topological embeddings $\mathcal{E}_1(G)$ and $\mathcal{E}_2(G)$ of a graph $G$ which correspond to the same combinatorial embedding $\hat{G}$ of $G$, there exists a bijection between the faces of $\mathcal{E}_1(G)$ and $\mathcal{E}_2(G)$ so that two corresponding faces have the same boundary. Hence, we can define the faces of $\hat{G}$ as the faces of any topological embedding consistent with $\hat{G}$.

Since a combinatorial embedding is sufficient for most graph algorithms, we restrict our attention to combinatorial embeddings only and refer to them simply as embeddings in the rest of this thesis.

### 3.2.7 Graph Separators and Graph Partitions

We call a set $S \subseteq V$ of vertices an *$\varepsilon$-vertex separator* of a weighted graph $G = (V, E)$, for $0 < \varepsilon < 1$, if no connected component of $G - S$ has weight more than $\varepsilon\omega(G)$. Similarly, an *$\varepsilon$-edge separator* of $G$ is a set $S \subseteq E$ of edges so that no connected component of graph $G' = (V, E \setminus S)$ has weight exceeding $\varepsilon\omega(G)$. Since we are interested mainly in vertex separators in this thesis, we refer to vertex separators simply as separators. If $G$ is unweighted, we assume that every vertex of $G$ has weight one.

It is a well-known fact that every tree has a $\frac{2}{3}$-separator of size one. Every outerplanar graph has a $\frac{2}{3}$-separator of size two. Lipton and Tarjan [124] show that every planar graph $G = (V, E)$ has a $\frac{2}{3}$-separator of size $\mathcal{O}\left(\sqrt{N}\right)$, where $N = |V|$. Recursive application of these results implies that every tree or outerplanar graph has an $\varepsilon$-separators of size $\mathcal{O}(1/\varepsilon)$, for any $0 < \varepsilon < 1$, and every planar graph has an $\varepsilon$-separator of size $\mathcal{O}\left(\sqrt{N/\varepsilon}\right)$.

Let $h > 0$ be an integer, $\varepsilon = h/N$, and $S$ be an $\varepsilon$-separator of a graph $G$. Let $H_1, \ldots, H_q$ be the connected components of $G - S$. Then an $h$-partition of $G$ is a pair $\mathcal{P} = (S, \{G_1, \ldots, G_r\})$ with the following properties:

(i) $\bigcup_{i=1}^{r} G_i = G - S$,

(ii) $G_i \cap G_j = \emptyset$, for all $1 \leq i, j \leq r$, $i \neq j$,

(iii) For all $1 \leq i \leq q$, there exists an index $1 \leq j \leq r$, so that $H_i \subseteq G_j$, and

(iv) $|V(G_i)| \leq h$, for all $1 \leq i \leq r$.

Intuitively, every graph $G_j$ is obtained by merging a number of connected components of $G - S$. For every graph $G_j$, $1 \leq j \leq r$, we define the *boundary* of $G_j$ as the set $\partial G_j \subseteq S$ of separator vertices that are adjacent to vertices in $G_j$.

If $G$ is a planar graph, we call an $h$-partition $\mathcal{P} = (S, \{G_1, \ldots, G_r\})$ *normal* if $r = \mathcal{O}(N/h)$ and $\sum_{i=1}^{r} |\partial G_i| = \mathcal{O}\left(N/\sqrt{h}\right)$. A normal $h$-partition $\mathcal{P} = (S, \{G_1, \ldots, G_r\})$ is *c-proper*, for some constant $c$, if $|\partial G_i| \leq c\sqrt{h}$, for all $1 \leq i \leq r$. We call $\mathcal{P}$ *proper* if $\mathcal{P}$ is 1-proper. Finally, let $R_i = G[V(G_i) \cup \partial G_i]$, for all $1 \leq i \leq r$. Then we call an $h$-partition $\mathcal{P} = (S, \{G_1, \ldots, G_r\})$ *regular* if for all $1 \leq i \leq r$, one of the following conditions holds:

(i) Graph $R_i$ is connected, or

(ii) There are at most two indices $1 \leq h < j \leq r$, $i \notin \{h, j\}$, so that $R_i \cap R_h \neq \emptyset$ and $R_i \cap R_j \neq \emptyset$. In this case, $R_h$ and $R_j$ are connected.

It can be shown that every planar graph has a proper $h$-partition and every planar graph of bounded degree has a regular proper $h$-partition (see e.g. [73]).

### 3.2.8 Geometric Graphs and Geometric Spanners

We call an edge-weighted graph $G = (V, E)$ a *geometric graph* if each of its vertices has an associated location in $\mathbb{R}^d$, and every edge $e = \{v, w\} \in E$ has a real weight $\omega(e) = \text{dist}_p(v, w)$, where $\text{dist}_p(v, w)$ denotes the distance between points $v$ and $w$ in the $L_p$-metric. For two closed point sets $A$ and $B$, let $\text{dist}_p(A, B) = \min\{\text{dist}_p(a, b) : a \in A \text{ and } b \in B\}$. We denote the length of a line-segment $e$ in the $L_p$-metric by $\|e\|_p$.

For a point set $S \subset \mathbb{R}^d$, the *complete Euclidean graph* $\mathcal{E}(S)$ is the geometric graph $\mathcal{E}(S) = (S, \{\{p, q\} : p, q \in S\})$. The lengths of the edges in $\mathcal{E}(S)$ are measured in the $L_2$-metric. A *t-spanner* of point set $S$ is a $t$-spanner of $\mathcal{E}(S)$.

For a set $P$ of polygonal obstacles in the plane with vertex set $S$, we define the *visibility graph* $\mathcal{V}(P)$ as the subgraph of $\mathcal{E}(S)$ which contains an edge $\{p, q\}$ if and only if this edge does not intersect the interior of an obstacle in $P$. A *t-spanner* of $P$ is a $t$-spanner of $\mathcal{V}(P)$.

## 3.3 Problem Definitions

Since we consider breadth-first search, depth-first search, and the single source shortest path problem on different classes of graphs, we define these problems here and do not repeat these definitions when discussing our algorithms for solving these problems on different classes of graphs.

In order to define these problems, we have to introduce a few definitions related to spanning trees of connected graphs. Given a connected graph $G = (V, E)$ and a spanning tree $T = (V, F)$ of $G$, we call every edge in $F$ a *tree edge* and every edge in $E \setminus F$ a *non-tree edge* of $G$. If $T$ is rooted at some vertex $r$, we call a non-tree edge $\{v, w\}$ a *back-edge* if w.l.o.g. $v$ is an ancestor of $w$. Otherwise, we call edge $\{v, w\}$ a *cross-edge*. The *level* $\ell_T(v)$ of a vertex $v \in T$ is the number of edges in the unique path from $r$ to $v$ in $T$.

**Breadth-first search.** A *breadth-first spanning tree* or *BFS-tree* of a connected graph $G$ is a rooted spanning tree of $G$ so that for any edge $\{v, w\} \in G$, $|\ell_T(v) - \ell_T(w)| \leq 1$. In particular, there are no back-edges w.r.t. $T$ in $G$ if $G$ is simple. *Breadth-first search* or *BFS* is the problem of computing a BFS-tree of a given graph $G$.

**Depth-first search.** A *depth-first spanning tree* or *DFS-tree* of a connected graph $G$ is a rooted spanning tree of $G$ so that all non-tree edges in $G$ are back-edges w.r.t. $T$. *Depth-first search* or *DFS* is the problem of computing a DFS-tree of a given graph $G$.

**Single source shortest paths.** A *shortest path tree* of a connected edge-weighted graph $G$ with source $s$ is a spanning tree $T$ of $G$ rooted at vertex $s$ so that for every vertex $v \in G$, the path in $T$ from $s$ to $v$ is a shortest path from $s$ to $v$ in $G$. The *single source shortest path (SSSP)* problem is the problem of computing a shortest path tree for a given graph $G$ and a source vertex $s$.

# Chapter 4

# Previous Work

In this chapter, we give an overview of existing results in the areas of research we address. Section 4.1 discusses previous results on solving the graph problems we consider. Section 4.2 discusses existing results on computing spanners and proximity problems.

The results in every section are divided into sequential algorithms, parallel algorithms, and external memory results. Sequential algorithms are included mostly for historical completeness, and for the sake of comparison, as they do not lead to I/O-efficient solutions in general. The discussion of parallel algorithms does not distinguish between the different types of PRAMs (i.e., EREW, CREW, CRCW), as the different restrictions are irrelevant when these algorithms are simulated using the PRAM simulation of [43] (see Section 2.2.2). Given the general analysis of the I/O-efficiency of PRAM algorithms provided in Section 2.2.2, we do not explicitly state the I/O-efficiency of the PRAM algorithms we discuss, unless the number of processors used and the amount of data processed in each step of the algorithm is geometrically decreasing.

Similarly, a sequential algorithm running in $\mathcal{O}(T(N))$ time can be assumed to take $\mathcal{O}(T(N))$ I/Os unless stated otherwise.

# 4.1 Graph Algorithms

## 4.1.1 Graph Searching

Breadth-first search (BFS) and depth-first search (DFS) are two fundamental techniques for systematically exploring a graph. Both techniques can be realized in linear time by very simple algorithms; yet they provide valuable information about the given graph. Most notably, all existing separator algorithms are based on breadth-first search. Depth-first search has been used to derive linear-time algorithms for a number of fundamental problems such as connectivity, biconnectivity [163], and triconnectivity [97] of graphs, as well as planar embedding [98]. Unfortunately, the vertex access patterns of these two search strategies are inherently sequential and seem to be inherently random. Reif [146] shows that ordered DFS is $P$-complete, thus not admitting any algorithm that solves this problem in polylogarithmic time with a polynomial number of processors, unless every problem solvable in polynomial time can be solved in polylogarithmic time using a polynomial number of processors. No truly efficient parallel or I/O-efficient algorithms are known for these two problems on general graphs. We discuss important previous results next.

When trying to compute a BFS-tree of a given graph in parallel, the problem does not appear much simpler than solving the more general single source shortest path problem on the given graph. Consequently, not much work has been done on computing BFS-trees in parallel. Ghosh and Bhattacharjee [85] present a parallel algorithm that computes a BFS-tree of an arbitrary graph in $\mathcal{O}(\log|V|\log d)$ time using $\mathcal{O}(|V|^3)$ processors, where $d$ is the diameter of the graph. In [109], the number of processors has been reduced to $\mathcal{O}(|V|^3/\log|V|)$. Gazit and Miller [84] present an algorithm for computing a BFS-tree of a given graph in $\mathcal{O}(\log^2|V|)$ time using $O(|V|^{2.376})$ processors. For shortest path algorithms see Section 4.1.2.

A number of parallel DFS-algorithms have been proposed. Using the same framework as for their BFS-algorithm, Kim and Chwa [109] obtain an $\mathcal{O}(\log|V|\log d)$ time algorithm which uses $\mathcal{O}(|V|^3/\log|V|)$ processors. Aggarwal and Anderson [4] present

a randomized algorithm which computes a DFS-tree for a given undirected graphs in $\mathcal{O}(T_{MM}(|V|)\log^3|V|)$ time using $\mathcal{O}(P_{MM}(|V|))$ processors, where $T_{MM}(|V|)$ and $P_{MM}(|V|)$ are the time and number of processors required to find a minimum weight perfect matching of a $|V|$-vertex graph with maximum edge weight $|V|$. In [5], the result has been generalized to directed graphs, leading to an algorithm that computes a DFS-tree for such graphs in $\mathcal{O}(\log^5|V|(T_{MM}(|V|)+\log^2|V|))$ time using $\mathcal{O}(P_{MM}(|V|)+M(|V|))$ processors, where $M(N)$ is the sequential time complexity for multiplying two $N\times N$ integer matrices. For planar graphs, a number of improved DFS algorithms have been proposed. The algorithm of Smith [162], though generally inefficient, provided the fundamental idea of using cycle separators for computing a DFS-tree of a planar graph. In [101, 156], $\mathcal{O}(\log|V|)$ time and linear processor algorithms for computing a cycle separator are presented, thereby leading to $\mathcal{O}(\log^2|V|)$ time DFS-algorithms, using the same number of processors.[1] He and Yesha [94] present another planar DFS-algorithm with the same time and processor bounds. In [105], the processor bound for finding a simple cycle separator has been reduced to $\mathcal{O}(|V|/\log|V|)$, thereby leading to an $\mathcal{O}(\log^2|V|)$ time DFS-algorithm for planar graphs which uses $\mathcal{O}(|V|/\log|V|)$ processors. Hagerup [92] presents an interesting idea of using BFS in the face incidence graph to derive a DFS-tree of an embedded planar graph. Given a BFS-tree of the face incidence graph, a DFS-tree of the given graph can be computed in $\mathcal{O}(\log|V|)$ time using $\mathcal{O}(|V|)$ operations. For directed planar graphs, Kao [104] presents a DFS algorithm which takes $\mathcal{O}(\log^5|V|)$ time using $\mathcal{O}(|V|/\log|V|)$ processors.

In external memory, the best known BFS-algorithm for undirected graphs is due to Munagala and Ranade [136] and takes $\mathcal{O}(|V|+\mathrm{sort}(|E|))$ I/Os. The best known DFS-algorithm for undirected graphs due to Kumar and Schwabe [118] takes $\mathcal{O}\left(\left(|V|+\frac{|E|}{B}\right)\log_2\frac{|V|}{B}+\mathrm{sort}(|E|)\right)$ I/Os. Chiang et al. [43] propose a DFS-algorithm

[1] In both papers [101, 156], the last case of the case analysis in finding the cycle separator is not handled correctly, and the produced separator may not be a simple cycle.

for directed graphs which takes $\mathcal{O}\left(|V| + \frac{|V|}{M}\text{scan}(|E|) + \text{sort}(|E|)\right)$ I/Os. Buchsbaum et al. [33] propose BFS and DFS-algorithms for directed graphs which take $\mathcal{O}\left(\left(|V| + \frac{|E|}{B}\right)\log_2 \frac{|V|}{B} + \text{sort}(|E|)\right)$ I/Os, thereby matching the DFS bound of [118] for the undirected case. In [132], Meyer shows that for any $0 < \gamma < 1/2$, the I/O-complexity of solving BFS on graphs of bounded degree can be reduced to $\mathcal{O}\left(\frac{|V|}{\gamma \log_d B} + \text{sort}(B^\gamma|V|)\right)$, at the expense of increasing the space used by the algorithm to $\mathcal{O}(B^\gamma|V|)$.

### 4.1.2 Shortest Path Problems on Graphs

The problem of computing shortest paths in graphs with real edge weights is well-studied. Shortest path problems can be divided into three groups according to the number of vertex pairs for which shortest paths are to be computed. The all pairs shortest path problem (APSP) is the problem of computing shortest paths for all pairs of vertices in the graph. The single source shortest path problem (SSSP) is the problem of computing shortest paths from a single vertex to all other vertices in the graph. Another important problem is that of finding a shortest path between two query vertices. Without preprocessing, however, this problem is no easier than the single source shortest path problem.

Note that a shortest path between two vertices $v$ and $w$ exists only if the graph does not contain a negative cycle which contains a vertex on a path from $v$ to $w$. This is trivially true if all edges have non-negative weights. In this case, Dijkstra's algorithm [60], when implemented using Fibonacci heaps [79], solves the single source shortest path problem in $\mathcal{O}(|V|\log_2|V| + |E|)$ time. If the graph contains negative edges, the Bellman-Ford algorithm [23, 72] solves the single source shortest path problem in $\mathcal{O}(|V||E|)$ time. The all pairs shortest path problem can be solved in $\mathcal{O}(|V|^3)$ time using the Floyd-Warshall algorithm [71, 173]. For sparse graphs, Johnson [103] presents an APSP algorithm which takes $\mathcal{O}(|E||V| + |V|^2\log_2|V|)$ time.

For graphs with non-negative integer weights, Ahuja et al. [7] propose an SSSP-algorithm that takes $\mathcal{O}\left(|E| + |V|\sqrt{\log_2 W}\right)$ time, where $W$ is the maximal edge

weight. Gabov and Tarjan [81] present an algorithm that solves the single source shortest path problem in $\mathcal{O}\left(\sqrt{|V|}|E|\log_2(|V|W)\right)$ time in the presence of negative edge weights, where $W$ is the absolute value of the edge weight of largest magnitude in the graph. Thorup [164] presents a linear-time single source shortest path algorithm for graphs with integer edge weights. In [165], the result is generalized to graphs with edge weights which are floats.[2]

For planar and outerplanar graphs, more efficient shortest path algorithms have been developed. Frederickson [73] proposes an algorithm, based on a recursive separator decomposition of the graph, which takes $\mathcal{O}\left(|V|\sqrt{\log_2|V|}\right)$ time to solve the single source shortest path problem on planar graphs with non-negative edge weights. In the same paper, an $\mathcal{O}(|V|^2)$ time algorithm to solve the APSP problem on planar graphs is presented. This algorithm is optimal if the output is to be represented as the distances between all pairs of vertices. In a later paper [74], two improved algorithms are presented which solve the APSP problem in $\mathcal{O}(q|V|)$ and $\mathcal{O}(|V|+q^2)$ time, where $q$ is the size of the smallest face-on-vertex cover. That is, two data structures are constructed, which allow answering shortest path queries in $\mathcal{O}(L\log N)$ time, where $L$ is the length of the reported path. Both algorithms are based on a "hammock decomposition" of the graph, which has later been applied to solve a number of other problems. In [62], an improved APSP algorithm based on a hammock decomposition is presented. The algorithm takes $\mathcal{O}(|V|\log|V|+q^2)$ time to build a data structure which can answer shortest path queries in $\mathcal{O}(\log N + L)$ time. The search for fast shortest path algorithms for planar graphs culminated in the linear time algorithm of [111]. As the algorithm of [73], it is based on a separator decomposition of the graph, but uses a rather complicated data structure consisting of a hierarchy of priority queues to achieve linear running time. For the case of negative edge weights, an $\mathcal{O}(N^{4/3}\log(NW))$ time algorithm for the single source shortest path problem is

[2]Floats are the numbers used in existing machines to approximate real numbers. The algorithm uses the fact that both integers and floats are represented as machine words; only the interpretation of the bits in the word is different.

presented in [111]. Here, $W$ denotes the largest absolute value of the weights of edges with negative weights.

The single source shortest path problem can be solved on outerplanar graphs and graphs of bounded treewidth in linear time, by applying dynamic programming to a tree-decomposition of the graph. In [76], Frederickson shows that an outerplanar graph can be preprocessed in linear time so that shortest path queries between any two vertices can be answered in $\mathcal{O}(\log N + L)$ time. In [65], the time for distance queries has been reduced to $\mathcal{O}(\log N)$. The query time has been reduced even further, to $\mathcal{O}(\alpha(N))$, in [75].

In the PRAM model, it is still one of the fundamental open problems to design truly efficient shortest path algorithms. Existing algorithms either use transitive closure computations (e.g., matrix multiplication) or exploit the structure exhibited by special classes of graphs. The former are highly inefficient in terms of the work they perform. The latter work only for restricted classes of graphs. The obvious algorithm that comes to mind to solve the all pairs shortest path problem is powering the distance matrix defined by the edge lengths $\log N$ times. This takes matrix multiplication, i.e., $\mathcal{O}(N^3)$ processors and $\mathcal{O}(\log^2 N)$ time. For the single source shortest path problem, there is no obvious way to parallelize it other than essentially solving all pairs shortest paths. For sparse graphs with small separators, Pan and Reif [139] present improved algorithms for the single source shortest path and the all pairs shortest path problems. Their SSSP algorithm takes $\mathcal{O}\left(\sqrt{|V|}\log|V|\right)$ time if no separator decomposition is given. Otherwise, the algorithm takes $\mathcal{O}(\log^2|V|)$ time on a CRCW PRAM and $\mathcal{O}(\log^3|V|)$ time on an EREW PRAM. The work performed by the algorithm is $\mathcal{O}(|V|^{3/2})$. For the all pairs shortest path problem, they present an $\mathcal{O}(|V|^2\log|V|)$ work algorithm which takes $\mathcal{O}\left(\sqrt{|V|}\log|V|\right)$ time if no separator decomposition is given, and $\mathcal{O}(\log^2 N)$ or $\mathcal{O}(\log^3 N)$ time otherwise, depending on whether a CRCW or EREW PRAM is used. Klein and Sairam [113] present a randomized algorithm which solves the approximate single source shortest path problem in $\mathcal{O}\left(\sqrt{|V|}\varepsilon^{-2}\log|V|\log^*|V|\right)$ time using $\mathcal{O}((|E|\log|V|)/\varepsilon^{-2})$ processors, with high

probability. Distances are approximated within a factor of $1+\varepsilon$ from their true values. For planar graphs, they give an improved algorithm which uses the same number of processors, but reduces the running time to $\mathcal{O}(|V|^{1/3}\varepsilon^{-2}\log|V|\log^*|V|)$. Cohen [48] presents an algorithm that solves the approximate multi-source shortest path problem in polylogarithmic time and performs $\mathcal{O}(|E||V|^{\alpha}+s(|E|+|V|^{1+\alpha}))$ work, where $s$ is the number of sources, and $\alpha > 0$ is an arbitrary constant. The approximation factor is $1+\mathcal{O}(1/\operatorname{polylog}|V|)$. For graphs with small separators, Cohen [47] presents a multi-source shortest path algorithm which takes polylogarithmic time and performs $\mathcal{O}(|V|^{3\mu}+s(|V|+|V|^{2\mu}))$ work, where $\mu$ is a constant $0 < \mu < 1$ so that every graph with $k$ vertices which is in the considered class has a separator of size $k^{\mu}$. For a planar graph and a single source, for example, the algorithm performs $\mathcal{O}(|V|^{3/2})$ work. The algorithm assumes that a separator decomposition of the graph is given as part of the input. For sparse graphs whose edges have non-negative integer weights, Klein and Subramanian [114] present a randomized algorithm that solves the single source shortest path problem in $\mathcal{O}(\operatorname{polylog}(|E|+L))$ time using $\mathcal{O}(|E|^2)$ processors, where $L$ is the maximal edge length in the graph. For planar graphs, they present an improved algorithm which uses $\mathcal{O}(|V|)$ processors. Träff and Zaroliagis [166] present a simple single source shortest path algorithm for planar directed graphs which takes $\mathcal{O}((N^{2\alpha}+N^{1-\alpha})\log N)$ time and performs $\mathcal{O}(N^{1+\alpha}\log N)$ work. The algorithm is based on a parallelization of Frederickson's algorithm [73]. Replacing Dijkstra's algorithm with the algorithm of [111] in the sequential shortest path computation, the work can be reduced to $\mathcal{O}(N^{1+\alpha})$ [166]; but the simplicity of the algorithm is lost. Pantziou, Spirakis, and Zaroliagis [140] present an algorithm for preprocessing a planar graph so that distance and shortest path queries can be answered in $\mathcal{O}(\log|V|)$ and $\mathcal{O}(\log|V|+L)$ time, respectively, where $L$ is the number of edges in the reported path. The algorithm takes $\mathcal{O}(\log^2|V|)$ time using $\mathcal{O}(q|V|+M(q))$ processors, where $M(k)$ is the number of processors required to multiply two $k\times k$ matrices, and $q$ is the number of faces in a given face-on-vertex cover of the graph. In [62], the processor bound has been improved to $\mathcal{O}(|V|+M(q))$,

using an $\mathcal{O}(|V|)$ preprocessing algorithm for outerplanar graphs as a building block. The results of [62, 140] are based on a parallelization of the hammock decomposition pioneered by Frederickson [73]. In [106], this decomposition has been generalized to arbitrary graphs, thereby allowing any graph to be decomposed into a small number of outerplanar graphs, depending on the structure of the given graph. Every sparse graph can be decomposed into $\mathcal{O}(|V|)$ outerplanar graphs. This decomposition is obtained in $\mathcal{O}(\log|V|\log\log|V|)$ time using $\mathcal{O}(|V|+|E|)$ processors on a CREW PRAM. If a CRCW PRAM is used, the time bound becomes $\mathcal{O}(\log|V|)$. The authors of [106] use this decomposition to design an algorithm that preprocesses a given graph in $\mathcal{O}(\log^2|V|)$ time using $\mathcal{O}(\gamma|V|+M(\gamma))$ processors and $\mathcal{O}(\gamma|V|)$ space for fast shortest path queries; but they do not state the query bound. Here, $\gamma$ denotes the number of outerplanar graphs into which the given graph has been decomposed. Similar improvements as for planar graphs allow the processor bound to be reduced to $\mathcal{O}(|V|+M(\gamma))$. For planar graphs, the processor bound can be reduced to $\mathcal{O}(\gamma|V|)$ or $\mathcal{O}(|V|+\gamma^2/\log^5\gamma)$ at the expense of increasing the running time to $\mathcal{O}(\log^2|V|+\log^5\gamma)$.

In external memory, the best known algorithm for the SSSP problem in graphs with non-negative edge weights is due to Kumar and Schwabe [118]. Their algorithm takes $\mathcal{O}(|V|+(|E|/B)\log_2|E|)$ I/Os. It is an implementation of Dijkstra's algorithm using an external tournament tree as priority queue. As with the BFS-algorithm of [136], the algorithm is efficient for dense graphs, but performs poorly on sparse graphs. In [52], a randomized single source shortest path algorithm is presented which takes $\mathcal{O}(|V|/D+\mathrm{sort}(|E|))$ I/Os with high probability on regular directed random graphs. For planar graphs of bounded degree, Arge, Brodal, and Toma [12] show that the single source shortest path problem can be solved in $\mathcal{O}(\mathrm{sort}(N))$ I/Os, provided that a regular proper $(DB)^2$-partition of the graph is given.

### 4.1.3 Planarity Testing and Planar Embedding

In the last few decades it has been shown that planar graphs and outerplanar graphs allow more efficient solutions to a number of graph problems than the general solutions for arbitrary graphs. Almost all these algorithms exploit the topological information provided by a planar or outerplanar embedding of the graph. Thus, it is an important problem to decide whether a given graph is planar or outerplanar and if so, compute a planar or outerplanar embedding of the graph.

The search for efficient planarity testing algorithms culminated in the linear time algorithm of Hopcroft and Tarjan [98]. An earlier algorithm of Lempel, Even, and Cederbaum [122] has later been made to run in linear time using techniques from [30, 70]. In [44], Chiba et al. give the details of using the algorithm of [30] to obtain a planar embedding of the given graph. Mehlhorn and Mutzel [131] provide important implementation details of the embedding phase of Hopcroft and Tarjan's algorithm. In [31], Boyer and Myrvold present a very elegant linear-time algorithm for planarity testing and planar embedding. The algorithm is much simpler than all previous algorithms for this problem.

In the PRAM model, Klein and Reif [112] present a planar embedding algorithm which runs in $\mathcal{O}(\log^2 N)$ time using $\mathcal{O}(N)$ processors, thereby improving on previous results of [102, 133]. As the algorithm of [30], it uses PQ-trees to maintain the set of valid planar embeddings of the graph, which is reduced step-by-step using careful modifications of the tree as more constraints in the graph are discovered. In [145], Ramachandran and Reif present a parallel algorithm to compute a planar embedding of a planar graph in $\mathcal{O}(\log N)$ time using the same number of processors as required for finding the connected components of a graph and performing bucket sort.[3]

[3]Contrary to a claim made in [145], we believe that the algorithm cannot be used to test whether a given graph is planar.

### 4.1.4 Graph Separators

A separator of a graph is a subset of the vertices (or edges) of the graph whose removal partitions the graph into subgraphs with few vertices. Separators are interesting from an algorithmic point of view, as they can be used to design efficient divide-and-conquer algorithms. If the number of vertices in the separator is small, the number of interactions between the different subgraphs is limited, so that the split or join phases of the algorithm become efficient.

It is well-known that every tree has a $\frac{2}{3}$-vertex separator of size one. This immediately implies the existence of a $\frac{2}{3}$-vertex separator of size two for outerplanar graphs. Lipton and Tarjan [124] were the first to show that every planar graph has a $\frac{2}{3}$-vertex separator of size $\mathcal{O}\left(\sqrt{N}\right)$. They also present a linear time algorithm to compute such a separator. In [86], the result is generalized to embedded graphs of bounded genus. In particular, the paper presents a linear time algorithm to compute a $\frac{2}{3}$-vertex separator of size $\mathcal{O}\left(\sqrt{gN}\right)$, where $g$ is the genus of the graph. Frederickson [73] applies the result of [124] recursively to compute a $c$-proper $h$-partition of a planar graph in $\mathcal{O}(N \log N)$ time. Aleksandrov and Djidjev [8] show how to compute such a partition in linear time if the constraint on the number of graphs and the boundary size of each individual component is dropped. Their result generalizes to embedded graphs of bounded genus. Other results include results on computing edge separators [61], vertex separators with negative and multiple vertex weights [64], and separators with vertex costs and weights [63]. Miller [134] presents a linear time algorithm to find a simple cycle separator of size $\mathcal{O}(\sqrt{fN})$ for a biconnected planar graph whose largest face has $f$ edges on its boundary.

A number of algorithms for computing separators in parallel have been proposed. Gazit and Miller [83] present a randomized parallel algorithm to compute a $\frac{2}{3}$-vertex separator of a planar graph in $\mathcal{O}(\log^2 N)$ time using $\mathcal{O}(N + F^{1+\varepsilon})$ processors, where $F$ is the number of faces in the graph. Using random-sampling, Klein [110] improves the work bound of the algorithm to $\mathcal{O}(N \log^2 N)$ at the expense of increasing the

running time to $\mathcal{O}(N^\varepsilon)$, for any $\varepsilon > 0$. Goodrich [88] gives a parallel and generalized version of Lipton and Tarjan's result, showing that it takes $\mathcal{O}(\log N)$ time using $\mathcal{O}(N/\log N)$ processors to compute a separator of size $\mathcal{O}(N^{1/2+\varepsilon})$ that divides a given planar graph into $\mathcal{O}(N^\varepsilon)$ subgraphs of size $\mathcal{O}(N^{1-\varepsilon})$. The algorithm requires a BFS-tree and an embedding of the graph to be given as part of the input. In some applications, including DFS, it is desirable that the computed separator have a particularly simple structure. In [134], Miller presents an algorithm to compute a simple cycle $\frac{2}{3}$-separator of size $\mathcal{O}\left(\sqrt{fN}\right)$ for an embedded biconnected planar graph, where $f$ is the size of the largest face of the graph. Provided that an embedding and a BFS-tree of the graph is given, the algorithm takes $\mathcal{O}(\log N)$ time using $\mathcal{O}(N)$ processors. In the case of depth-first search, the size of the separator is not important. For this case, $\mathcal{O}(\log^2 N)$ time and linear processor algorithms are given in [94, 101, 156]. In [105], the running time has been reduced to $\mathcal{O}(\log N)$ using $\mathcal{O}(N/\log N)$ processors, thereby achieving an optimal linear work bound.

The first to present an I/O-efficient algorithm for computing graph separators were Hutchinson, Maheshwari, and Zeh [100]. Their algorithm takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os to compute a $\frac{2}{3}$-vertex separator of size $\mathcal{O}\left(\sqrt{N}\right)$ for an embedded planar graph, provided that a BFS-tree of the graph is part of the input. The algorithm is an I/O-efficient variant of the algorithm of Lipton and Tarjan [124]. Arge, Brodal, and Toma [12] generalize the approach of [100] to obtain an $\varepsilon$-vertex separator of size $\mathcal{O}\left(\mathrm{sort}(N) + \sqrt{N/\varepsilon}\right)$ for an embedded planar graph. The algorithm is based on the algorithm of [88] and takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, again assuming that a BFS-tree of the graph is given as part of the input.

## 4.2 Geometric Spanners and Proximity Problems

We conclude our survey of previous results with a discussion of results on computing geometric spanners and solving proximity problems. Eppstein [67] and Smid [160, 137] give excellent surveys of results in these areas. We only discuss the most

relevant results here and refer the reader to these publications for a more extensive discussion. In Sections 4.2.1 through Section 4.2.3, we review previous results on proximity problems. In Section 4.2.4, we discuss previous work on constructing geometric spanners.

### 4.2.1 Closest Pairs

The closest pair problem is that of finding the distance between the closest two points among the elements of a point set in $\mathbb{R}^d$. This problem can trivially be solved in $\mathcal{O}(N^2)$ time by examining all possible point pairs and reporting the shortest distance. But this leaves a gap to the $\Omega(N \log N)$ lower bound provable in the algebraic computation tree model [24]. The first optimal algorithms solving this problem in two dimensions are due to Shamos [154] and Shamos and Hoey [155]. In [95], Hinrichs, Nievergelt, and Schorn present a very elegant $\mathcal{O}(N \log N)$ time solution for the planar case, which is based on the plane-sweep paradigm. Using a divide-and-conquer approach, Bentley and Shamos [26] present an $\mathcal{O}(N \log N)$ solution in $\mathbb{R}^d$. However, the algorithm is rather complicated. Lenhof and Smid [123] present a rather simple and practical algorithm which solves the problem in $\mathcal{O}(N \log N)$ time. Their algorithm uses the floor function and indirect addressing, thereby not conforming with the algebraic model of computation. The use of the floor function can be avoided, but it is not clear how to avoid the use of indirect addressing in their algorithm. If the use of randomization and non-algebraic operations is allowed, the $\Omega(N \log N)$ lower bound can be beaten. Rabin [144] presents an algorithm that runs in expected linear time when implemented using the perfect hashing scheme of Fredman, Komlós and Szemerédi [78]. Another algorithm which takes expected linear time is proposed in [108]. The algorithm is based on sieving out points whose closest neighbors are too far away. Golin et al. [87] propose an algorithm which is based on the randomized incremental construction paradigm [153]. The algorithm takes expected linear time and $\mathcal{O}\left(N \frac{\log N}{\log \log N}\right)$ time with high probability.

The closest pair problem can be solved in the PRAM model using the all nearest neighbor algorithms discussed in Section 4.2.3.

In external memory, the closest pair of a planar point set can be computed in optimal $\mathcal{O}(\text{sort}(N))$ I/Os using the all nearest neighbor algorithm of [90]. The optimality of this algorithm is shown in [14]. It can also be computed in the same I/O-bound using the Voronoi diagram construction of [51], deriving the Delaunay triangulation from the Voronoi diagram, and then scanning the resulting set of edges. Using this approach the I/O-bound is expected, as the algorithm in [51] is randomized.

### 4.2.2 $K$-Closest Pairs

An obvious extension of the closest pair problem is that of reporting the $K$ closest pairs instead of only the closest pair. For this problem, the first two algorithms, due to Smid [159], spend $\mathcal{O}\left(N \log N + N\sqrt{K} \log K\right)$ time in two dimensions and $\mathcal{O}\left(N^{4/3} \log N + N\sqrt{K} \log K\right)$ time in $d$ dimensions, $d > 2$. Dickerson, Drysdale, and Sack [58] give a simple $\mathcal{O}(N \log N + K \log K)$ time algorithm for the planar case. In [59], Dickerson and Eppstein present a non-trivial extension of the result of [58] to higher dimensions, which achieves the same running time as the algorithm of [58]. If the distances do not need to be reported in sorted order, an improved $\mathcal{O}(N \log N + K)$ time algorithm is presented in [59]. Salowe [151] presents another $\mathcal{O}(N \log N + K)$ time algorithm for this problem, which uses the algorithm of [168] and parametric search. Lenhof and Smid [123] give a much simpler algorithm achieving the same running time; but they use indirect addressing, thereby leaving the algebraic model of computation. The well-separated pair decomposition introduced by Callahan and Kosaraju [37] can also be used to compute the $K$ closest pairs in optimal $\mathcal{O}(N \log N + K)$ time in $d$ dimensions. For details, see [35].

In the PRAM model, Lenhof and Smid [123] propose a randomized algorithm which runs in $\mathcal{O}(\log^2 N \log\log N)$ expected time using $\mathcal{O}(N \log N \log\log N + K)$ work. The suboptimality of the algorithm and the randomization stem from the use of the best known integer sorting algorithm at that time [130], which is randomized and takes

$\mathcal{O}(\log N \log\log N)$ time using $\mathcal{O}(N \log\log N)$ work. Using a more recent result [89], the time and work bounds can be reduced to $\mathcal{O}(\log^2 N)$ and $O(N \log N + K)$, respectively. Again, the bounds are expected. Even though not shown by Callahan and Kosaraju, we believe that their $K$-closest pair algorithm [38] can be parallelized to compute the $K$ closest pairs in $\mathcal{O}(\log N)$ time performing $\mathcal{O}(N \log N + K)$ work.

### 4.2.3 All Nearest Neighbors

Another extension of the closest pair problem is that of computing the nearest neighbor, for every point in the set. This problem is also known as the all nearest neighbor problem. The algorithms of [155] can easily be extended to compute all nearest neighbors in two dimensions in optimal $\mathcal{O}(N \log N)$ time. In [96], Hinrichs, Nievergelt, and Schorn give an extension of their closest pair algorithm, which solves the all nearest neighbor problem in the plane in $\mathcal{O}(N \log N)$ time. Bentley [25] shows how to extend his closest pair algorithm for the $d$-dimensional case so that it can solve the all nearest neighbor problem in $\mathcal{O}(N \log^{d-1} N)$ time. The first $\mathcal{O}(N \log N)$ time algorithm to solve the all nearest neighbor problem in higher dimensions is due to Clarkson [45]. The algorithm is randomized. Hence, the running time is expected. Vaidya [168] gives the first deterministic $\mathcal{O}(N \log N)$ time algorithm for this problem. Callahan and Kosaraju [39] show that their well-separated pair decomposition can also be used to compute all nearest neighbors in $\mathcal{O}(N \log N)$ time. In fact, they show that the more general problem of computing the $K$ nearest neighbors for every point in the point set can be solved in $\mathcal{O}(N \log N + KN)$ time.

In the PRAM model, Cole and Goodrich [49] present an $\mathcal{O}(\log N)$ time and linear processor algorithm to solve the all nearest neighbor problem. Frieze, Miller, and Teng [80] present a randomized algorithm to solve the $K$-nearest neighbor problem in $d$ dimensions in expected $\mathcal{O}(\log N)$ time using $\mathcal{O}(N)$ processors, provided that $d$ and $K$ are fixed. Callahan and Kosaraju [34, 39] show that the well-separated pair decomposition can be used to solve the $K$-nearest neighbor problem in $\mathcal{O}(\log N)$ time using $\mathcal{O}(KN)$ processors.

In external memory, the only all nearest neighbor algorithm that we are aware of is that of [90], which solves the problem in $\mathcal{O}(\text{sort}(N))$ I/Os in two dimensions.

### 4.2.4 Geometric Spanners

As discussed in Section 3.2.8, a geometric spanner is a spanner graph that approximates the complete Euclidean graph of a point set or the visibility graph of a set of polygonal obstacles so that the distances between points or obstacle vertices are preserved up to a constant factor. Besides the spanning ratio $t$, other optimization criteria include the size, the spanner diameter, the total weight (i.e., the sum of the lengths of all edges in the graph), and the degree of the vertices in the spanner. All the spanners discussed here have $\mathcal{O}(N)$ edges.

Even though the minimum spanning tree of a point set does not exhibit a good spanning ratio, it is important in the context of computing spanners, as it provides a lower bound on the weight of any spanner. The minimum spanning tree in the plane can be computed in $\mathcal{O}(N \log N)$ time for instance by computing the Delaunay triangulation [141] and then applying any standard minimum spanning tree algorithm [117, 142] for graphs to the Delaunay triangulation. In general, any sparse graph which is guaranteed to contain the minimum spanning tree as a subgraph can be used instead of the Delaunay triangulation. It has been shown by Shamos and Hoey [155] that the minimum spanning tree of a point set is a subgraph of the Delaunay triangulation of the point set. In higher dimensions, finding such a sparse graph is harder than in the plane. Yao [176] presents an $\mathcal{O}(N^{2-\varepsilon_d})$ time algorithm for finding a minimum spanning tree in higher dimensions, where $\varepsilon_d$ is a small constant depending on the dimension $d$. The next improvement has been achieved by Agarwal et al. [3], who show that the problem can be solved in $\mathcal{O}\left((N \log N)^{4/3}\right)$ time, for $d = 3$, and $\mathcal{O}\left(N^{2-2/(\lceil d/2 \rceil + 1)+\varepsilon}\right)$ time, for $d > 3$ and any $\varepsilon > 0$. Callahan and Kosaraju [39] show how to compute an approximate minimum spanning tree of weight at most $1+\varepsilon$ times the weight of the minimum spanning tree. Their algorithm takes $\mathcal{O}(N \log N)$ time. Similar results have been achieved in [150, 167].

The concept of spanner graphs has been introduced by Chew [42], who shows that the rectilinear Delaunay triangulation has spanning ratio $\sqrt{10}$. He also shows that the Euclidean Delaunay triangulation has spanning ratio at least $\pi/2$ in the worst case. Dobkin et al. [66] show that the Euclidean Delaunay triangulation has spanning ratio at most $\frac{(1+\sqrt{5})\pi}{2} \approx 5.08$. Keil and Gutwin [107] prove that the spanning ratio is in fact no more than $\frac{2\pi}{3\cos(\pi/6)} \approx 2.42$. Unfortunately, by the result of [42], the Delaunay triangulation cannot be used when a spanning ratio arbitrarily close to one is desired. In fact, it is easy to show that there is no planar graph which has spanning ratio less than $\sqrt{2}$ in the worst case.

The first to show how to construct a $t$-spanner in the plane, for $t$ arbitrarily close to one, were Keil and Gutwin [107]. Independently, Clarkson [46] discovered the same construction for $d = 2$ and $d = 3$. Ruppert and Seidel [148] generalize the result to higher dimensions, using the construction of a $\theta$-frame due to Yao [176]. The algorithm of [148] takes $\mathcal{O}\left(N \log^{d-1} N\right)$ time. Arya, Mount, and Smid [19] combine the $\theta$-graph of Ruppert and Seidel with skip lists [143] to obtain a $t$-spanner which has $\mathcal{O}(N)$ edges and spanner diameter $\mathcal{O}(\log N)$, with high probability. Arya et al. [18] show how to transform any spanner with bounded out-degree into a spanner of bounded in- and out-degree. This construction takes $\mathcal{O}(N \log N)$ time, thus leading to an $\mathcal{O}\left(N \log^{d-1} N\right)$ time algorithm to construct a bounded degree spanner based on the $\theta$-graph. Another bounded degree spanner is the greedy spanner which is obtained by examining the edges of the complete Euclidean graph sorted by increasing length and adding an edge to the spanner if the current graph does not contain a path between its endpoints which is at most $t$ times their Euclidean distance. The fact that this graph has bounded degree has been shown by Chandra et al. [41]. Results by Das, Heffernan, and Narasimhan [53] and Das, Narasimhan, and Salowe [55] show that the weight of the greedy spanner is proportional to that of a minimum spanning tree of the point set. In order to reduce the construction time of the greedy spanner from $\Omega(N^2)$ to $\mathcal{O}\left(N \log^2 N\right)$, Das and Narasimhan [54] propose an algorithm based on graph clustering techniques. The resulting graph has weight proportional to that

of a minimum spanning tree; but its degree may be large. Arya and Smid [20] present an $\mathcal{O}\left(N \log^d N\right)$ time algorithm to compute a $t$-spanner of bounded degree whose weight is bounded by $\mathcal{O}(\log N)$ times the weight of a minimum spanning tree. By applying the results of [54] to this spanner, one obtains an $\mathcal{O}\left(N \log^d N\right)$ time algorithm to compute a $t$-spanner of bounded degree and weight within a constant factor of that of a minimum spanning tree. Vaidya [169] and Salowe [150] were the first to give optimal $\mathcal{O}(N \log N)$ time algorithms for constructing $t$-spanners in higher dimensions. Their algorithms use hierarchical subdivisions similar to that induced by a fair split tree [39]. Consequently, the well-separated pair decomposition of [39] can also be used to obtain an $\mathcal{O}(N \log N)$ time algorithm for constructing $t$-spanners [38]. Arya, Mount, and Smid [19] show that the spanner obtained in this way can be constructed more carefully to guarantee that the spanner diameter is $\mathcal{O}(\log N)$. In [18], it is shown that the weight of this spanner is bounded by $\mathcal{O}(\log N)$ times the weight of a minimum spanning tree. The degree, however, is unbounded. An important result of [18] in terms of building data structures for efficiently reporting spanner paths is the following: They show that the WSPD-spanner can be augmented with a linear number of edges so that the resulting graph can be decomposed into a constant number of rooted trees with the property that for every pair of points, there exists a spanner path between them which stays completely inside one of these trees. Again, the spanner diameter of this spanner can be shown to be $\mathcal{O}(\log N)$ if the spanner is constructed carefully. By further augmenting the spanner with $\mathcal{O}(N)$ edges using standard shortcutting techniques for trees [10, 29], the diameter can be reduced to $\mathcal{O}(\alpha(N))$.

In [46], Clarkson shows that a modification of the $\theta$-graph for sets of polygonal obstacles is a $t$-spanner for the visibility graph of these obstacles. Arikati et al. [16] show how to construct a planar $t$-spanner among obstacles which contains only $\mathcal{O}(N)$ additional vertices, called *Steiner points*. Both constructions [16, 46] take $\mathcal{O}(N \log N)$ time.

Using their parallel algorithm for constructing a well-separated pair decomposition, Callahan and Kosaraju [34, 39] show that their approximate minimum spanning tree algorithm can be made to run in $\mathcal{O}(\log N)$ time using $\mathcal{O}(N)$ processors. Similarly, their spanner construction can be parallelized to obtain the same bounds.

In external memory, the only results related to computing spanning graphs are those of [90] and [51]. In these two papers it is shown how to compute the convex hull of a three-dimensional point set in $\mathcal{O}(\text{sort}(N))$ I/Os worst-case and expected, respectively. Using these algorithms, the Voronoi diagram of a two-dimensional point set can be computed, which in turn can be used to compute the Delaunay triangulation of the point set. Given the Delaunay triangulation, the minimum spanning tree algorithm for planar graphs of [43] can be applied to obtain the Euclidean minimum spanning tree of the point set. A randomized algorithm to compute the Voronoi diagram of a point set in the same number of I/Os without using convex hulls in $\mathbb{R}^3$ is also presented in [51].

# Part I

# Graph Algorithms

# Chapter 5

# Techniques for Solving Graph Problems

In this chapter, we develop a "toolchest" for solving graph problems I/O-efficiently. In particular, in Sections 5.1 and 5.2, we recall a number of standard data structures and paradigms for designing I/O-efficient graph algorithms. In Section 5.3, we define a number of primitive operations on graphs which can be realized in $\mathcal{O}(\text{sort}(N))$ I/Os. We make extensive use of these operations in our algorithms, so that their description is simplified if we have this set of primitives at our disposal.

## 5.1 Data Structures

We start our review with a discussion of I/O-efficient data structures that can be used to design I/O-efficient graph algorithms.

### 5.1.1 An I/O-Efficient Queue

A *queue* (e.g., see [50], Chapter 11) is a simple data structure to be used when an algorithm produces data that it has to process at a later point, and it wants to guarantee that the data is processed in the same order as it is produced. In

particular, a queue supports four operations: CREATE, ENQUEUE, DEQUEUE, and ISEMPTY. CREATE creates an empty queue. ENQUEUE appends its argument to the queue. DEQUEUE removes the least recently appended item from the queue and returns it. ISEMPTY tests whether the queue is empty.

Breadth-first search is a perfect example where a queue can be used. Initially, the source of the search is appended to the queue. Then the algorithm keeps removing vertices from the head of the queue. For every removed vertex $v$, it explores the neighbors of $v$, appends all neighbors which have not been visited before to the queue, and makes $v$ their parent in the computed BFS-tree.

In internal memory, a linked list is sufficient to implement a queue which supports all of the above operations in $\mathcal{O}(1)$ time. In order to make this implementation of a queue I/O-efficient, the queue is maintained as a linked list of "superblocks", each consisting of $D$ blocks, striped across the $D$ disks. Before appending a superblock to the end of the queue, the next $DB$ data items to be appended to the queue are inserted into an input buffer, which is held in internal memory. When the buffer runs full, its content is written to the next superblock of the queue. Similarly, when removing data from the queue, the first superblock is removed from the queue, and its content is loaded into an output buffer. The next $DB$ DEQUEUE operations read their respective data from the output buffer rather than the queue itself. This way, one I/O is spent per $DB$ ENQUEUE operations, and one I/O is spent per $DB$ DEQUEUE operations, so that a sequence of $N$ such operations can be processed in $\mathcal{O}\left(\frac{N}{DB}\right)$ I/Os.

### 5.1.2 An I/O-Efficient Stack

A *stack* (e.g., see [50], Chapter 11) is a data structure very similar to a queue. The only difference is that the data elements are removed from the data structure in reverse insertion order. That is, the data element that has been inserted last is removed first. Formally, a stack supports four operations: CREATE, PUSH, POP, and ISEMPTY. CREATE creates an empty stack. PUSH adds a new element to the

stack. POP removes the most recently added element from the stack and returns it. ISEMPTY tests whether the stack is empty.

Sticking to graph exploration as the illustrating example, the BFS-algorithm described above is easily turned into a DFS-algorithm by replacing the queue with a stack and each ENQUEUE and DEQUEUE operation with a PUSH and POP operation, respectively.

Again, a linked list is sufficient to implement a stack which supports all of the above operations in $\mathcal{O}(1)$ time. In external memory, this linked list is again maintained at the level of "superblocks", and the data structure is augmented with a buffer for inserting and removing elements. This time, however, it is sufficient to have only one buffer, acting as both input and output buffer, as data is inserted and removed at the same end of the list. On the other hand, the buffer needs to have size $2DB$, in order to guarantee that $\mathcal{O}(1)$ I/Os are spent per $DB$ stack operations. Otherwise, a sequence of stack operations where two PUSH operations alternate with two POP operations could force the algorithm to perform one I/O for every other operation: The first PUSH makes the buffer run full, so that an I/O is required to empty the buffer before the next PUSH can be performed. Then the first POP removes the element just pushed from the buffer, and the next POP operation necessitates an I/O-operation, in order for the next element to be read into internal memory.

With a buffer of size $2DB$, only $DB$ elements are written to disk when the buffer runs full, and only $DB$ elements are read from disk when the buffer becomes empty. In both cases, the buffer contains $DB$ data items after an I/O-operation, so that $DB$ PUSH operations are required to fill up the buffer, and $DB$ POP operation are required to empty the buffer. Hence, the next I/O is necessary after at least $DB$ stack operations.

### 5.1.3 The Buffer Tree—An I/O-Efficient Search Tree

The buffer tree [11] is an extension of the well-known $B$-tree data structure [22] which outperforms the $B$-tree in applications where a large number of updates and

queries need to be performed and immediate query responses are not required. For the sake of simplicity, we restrict our discussion of the buffer tree to the single disk case ($D = 1$). As the buffer tree can be seen as a data structure which makes a cascaded application of the distribution paradigm explicit, the same solutions that lead to I/O-optimal DISTRIBUTIONSORT algorithms on multiple disks [138, 172] can be used to make the buffer tree achieve optimal performance on multiple disks.

Intuitively, the buffer tree is a $B$-tree with increased fan-out whose nodes have been augmented with buffers to form batches of update and query operations to be processed w.r.t. the subtrees rooted at these nodes. That is, the buffer tree is an $(a, b)$-tree [99], where $a = m/4$ and $b = m$, $m = M/B$. Instead of processing an update or query operation immediately, it is appended to a buffer of size $B$, which is held in internal memory. When this buffer runs full, its content is appended to the buffer associated with the root of the buffer tree. If the root buffer contains less than $M/2$ data items and the algorithm is not finished yet, this finishes the operation. Otherwise, the root buffer is emptied by distributing its contents to the buffers of its children. If the buffer of a child runs full, this buffer is emptied recursively. The buffer emptying process loads the $m$ splitter elements as well as the first $M/2$ operations from the buffer into internal memory. This takes $\mathcal{O}(m)$ I/Os. Distributing the elements to the buffers of the children takes $\mathcal{O}(m)$ I/Os for writing complete blocks to the buffers of the children, as one block of data per child can be held in internal memory, and $\mathcal{O}(m)$ I/Os for excess I/Os, $\mathcal{O}(1)$ per child. Hence, distributing $M/2$ operations from a node to its children costs $\mathcal{O}(m)$ I/Os, $\mathcal{O}(1/B)$ I/Os amortized per distributed operation. Once the first $M/2$ elements in the buffer have been processed, the next $M/2$ elements are loaded into memory and the whole process is iterated until the buffer is empty.

At a leaf, instead of distributing the content of its buffer to those of its (non-existent) children, the operations in the buffer are processed in their order of appearance in the buffer. An INSERT operation adds a new data element to be stored at this leaf. A DELETE operation deletes a data item that is stored at this leaf. A SEARCH

operation tests whether the desired data item is stored at this leaf and reports it if this is the case. If an update operation causes a leaf to store more than $B$ data items or less than $B/4$ data items, the leaf has to be split or merged with its neighbors. This may cause a series of splits or merges along the path from the leaf toward the root. These structural changes to the tree are handled in a manner similar to $(a, b)$-trees. The analysis of the overall complexity of update operations is analogous to that for $(a, b)$-trees. (See [11] for details.) In total, every update or query operation in a sequence of $N$ updates or queries is charged for $\mathcal{O}(1/B)$ I/Os per level of the tree to account for I/Os spent on emptying buffers and rebalancing the tree. Since the fan-out of the tree is $\Theta(m)$, the height of the tree is $\mathcal{O}(\log_m(N/B))$, so that processing a sequence of $N$ updates and queries takes $\mathcal{O}\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) = \mathcal{O}(\text{sort}(N))$ I/Os.

### 5.1.4 An I/O-Efficient Priority Queue

In [11], a priority queue based on the buffer tree is proposed. This priority queue can process any sequence of $N$ Insert, Delete, and DeleteMin operations in $\mathcal{O}(\text{sort}(N))$ I/Os. Unfortunately, it does not support the DecreaseKey operation. If the DecreaseKey operation is needed, the I/O-efficient tournament tree of [118] has to be used, which is less efficient than the priority queue we describe here.

The basic idea proposed in [11] is to use the buffer tree to maintain the items in the priority queue sorted by their priorities. In a standard $(a, b)$-tree, it would be guaranteed that the element with the smallest priority is stored at the leftmost leaf of the tree. For the buffer tree, this is not true because there may be elements with smaller priorities stored along the path from the root to the leftmost leaf. However, before processing the first DeleteMin operation, one can perform forced buffer-emptying processes along the path from the root to the leftmost leaf. This takes $\mathcal{O}\left(\frac{M}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os for all buffers along the path. It may also trigger buffers of nodes that are not on the path to be emptied. The I/Os required to empty the latter buffers can be charged to Insert operations in the standard fashion, so that emptying the buffers along the leftmost path takes $\mathcal{O}\left(\frac{M}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os in an amortized sense.

Once these buffers have been emptied it is not only guaranteed that the leftmost leaf of the tree stores the element with the smallest priority, but that the children of the parent of the leftmost leaf store the elements with smallest priorities in the priority queue. Since every internal node of the buffer tree has at least $m/4$ children, and every leaf stores at least $B/4$ data items, the children of the parent of the leftmost leaf store at least $M/16$ elements. These elements are loaded into internal memory, so that at least $M/16$ DeleteMin operations can be performed without performing any I/Os. Hence, the amortized cost per DeleteMin operation is $\mathcal{O}\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os.

Once a number of smallest elements are stored in internal memory, subsequent Insert and Delete operations need to examine the elements stored in internal memory. This is to guarantee that no element is inserted which is smaller than the elements stored in internal memory, and none of the elements in internal memory is deleted from the priority queue before it is removed using a DeleteMin operation. This can be done without incurring any extra I/Os. Hence, Insert, Delete, and DeleteMin operations take $\mathcal{O}\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os in an amortized sense.

## 5.2 Paradigms and Techniques

In this section, we recall a number of paradigms and techniques that have been used to obtain I/O-efficient graph algorithm. The first of these paradigms, the data structuring paradigm, makes use of the data structures discussed in the previous section.

### 5.2.1 Data Structuring

As should be expected, algorithms based on the *data structuring* paradigm make use of I/O-efficient data structures to solve the problem at hand. However, the meaning of this approach in the context of the design of I/O-efficient algorithms is a little more drastic than in the context of algorithm design in general. In particular, a number of I/O-efficient algorithms have been obtained by taking an efficient internal memory

algorithm and replacing the data structures in this algorithm with their I/O-efficient counterparts. A prime example for this approach is the SSSP algorithm of Kumar and Schwabe [118], where the only modification of the algorithm is the replacement of the Fibonacci heap used in an efficient implementation of Dijkstra's algorithm with an I/O-efficient tournament tree.

### 5.2.2 Graph Contraction

A number of I/O-efficient graph algorithms are based on a paradigm that has been successfully applied to obtain efficient parallel graph algorithms: *graph contraction*. At a very abstract level, the paradigm is simple and elegant: Identify a number of disjoint subgraphs of $G$ so that representing each such subgraph by a single vertex reduces the size of $G$ by a constant factor and preserves the properties of interest. In parallel algorithms this compression technique is applied recursively $\mathcal{O}(\log N)$ times until the resulting graph has constant size. Then the problem is solved in $\mathcal{O}(1)$ time on the contracted graph. A solution for graph $G$ is constructed by undoing the contraction steps and at each step deriving a solution for the uncontracted graph from the given solution for the contracted graph. When designing I/O-efficient algorithms, the contraction can usually stop after $\mathcal{O}(\log DB)$ compression levels. At that point the resulting graph is guaranteed to have $\mathcal{O}\left(\frac{N}{DB}\right)$ vertices, so that the algorithm can afford to spend $\mathcal{O}(1)$ I/Os per vertex to solve the problem on the contracted graph. The edges can usually be handled using I/O-efficient data structures.

A simple example for the application of this paradigm is the computation of the connected components of a given graph. The connectivity of the given graph does not change if an edge of the graph is contracted. More generally, the connectivity of the graph does not change if connected subgraphs of $G$ are replaced by a single vertex each. The graph connectivity algorithm of [43] uses this fact to contract connected subgraphs of $G$ until every connected component consists of a single vertex. This provides a natural labelling of the connected components of $G$, and the vertices of $G$

can easily be labelled with their component labels when undoing the contraction steps.

In general, graph contraction does not lead to I/O-optimal algorithms. However, for sparse graphs under edge-contraction, a reduction of the number of vertices leads to a proportional reduction of the number of edges. Hence, if a constant fraction of the vertices are eliminated in each contraction step, the sizes of the graphs produced by repeated contraction are geometrically decreasing. Thus, if a contraction step can be realized in $\mathcal{O}(\text{sort}(|V|+|E|))$ I/Os, the whole algorithm takes $\mathcal{O}(\text{sort}(|V|+|E|))$ I/Os.

### 5.2.3 Time-Forward Processing

*Time-forward processing* is a very elegant technique for solving graph problems, which has been proposed in [43]. The boundary conditions for an I/O-efficient implementation of this technique have later been removed in [11]. The following problem can be solved using this technique:

Let $G$ be a directed acyclic graph whose vertices are numbered so that every edge in $G$ leads from a vertex with lower number to a vertex with higher number. Let every vertex $v$ of $G$ store a label $\phi(v)$, and let $f$ be a function to be applied in order to compute for every vertex $v$, a new label $\psi(v) = f(\phi(v), \lambda(u_1), \ldots, \lambda(u_k))$, where $u_1, \ldots, u_k$ are the in-neighbors of $v$ in $G$, and $\lambda(u_i)$ is some piece of information "sent" from $u_i$ to $v$ after computing $\psi(u_i)$. The goal is to "evaluate" $G$, i.e., to compute $\psi(v)$, for all vertices $v \in G$.

While time-forward processing does not solve the problem of computing $\psi(v)$ I/O-efficiently in the case where the input data $\phi(v)$ and $\lambda(u_1), \ldots, \lambda(u_k)$ do not fit into internal memory, it provides an elegant way to supply vertex $v$ with this information at the time when $\psi(v)$ is computed. The idea is to process the vertices in $G$ by increasing numbers. This guarantees that all in-neighbors of vertex $v$ have been evaluated at the time when $v$ is being evaluated. Thus, if these in-neighbors have "sent" their outputs $\lambda(u_1), \ldots, \lambda(u_k)$ to $v$, $v$ has these inputs and its own label $\phi(v)$ at its disposal to compute $\psi(v)$. After computing $\psi(v)$, $v$ sends its output $\lambda(v)$ "forward

in time" to its out-neighbors, which guarantees that these out-neighbors have all their inputs at their disposal when it is their turn to be evaluated.

**Time-forward processing for general DAGs.** The implementation of this technique due to Arge [11] is simple and elegant. The "sending" of information is realized using a priority queue $Q$. When a vertex $v$ wants to send its output $\lambda(v)$ to another vertex $w$, it inserts $\lambda(v)$ into priority queue $Q$ and gives it priority $w$. When vertex $w$ is being evaluated, it removes all entries with priority $w$ from $Q$. As every in-neighbor of $w$ has sent its output to $w$ by queuing it with priority $w$, this provides $w$ with the required inputs. Moreover, every vertex removes its inputs from the priority queue before it is evaluated, and all vertices with smaller numbers have been evaluated before $w$. Thus, the entries in $Q$ with priority $w$ are those with lowest priority, so that they can be removed using a sequence of DeleteMin operations.

Using the priority queue described in Section 5.1.4, evaluating graph $G$ takes $\mathcal{O}(\text{sort}(N+I))$ I/Os, where $N$ is the number of vertices in the graph, and $I$ is the total amount of information sent along the edges of $G$.

**Time-forward processing for rooted trees.** If the graph $G$ to be evaluated is a rooted tree, and its vertices are stored in preorder, an even simpler solution can be used to evaluate $G$ in a linear number of I/Os: Evaluate the nodes of $G$ in preorder, and use a stack $S$ to simulate the sending of data along the edges of $G$. In order to process $G$ from the root toward the leaves, every vertex pushes the outputs for its children on stack $S$, sorted in reverse preorder of the recipients. When a node is evaluated it retrieves its input from the top of the stack. To process $G$ from the leaves toward the root, reverse this procedure.

The number of I/O-operations performed by this algorithm is $\mathcal{O}(\text{scan}(N))$ for scanning the vertex list of $G$ and $\mathcal{O}(\text{scan}(N+I))$ for performing the stack operations required to send $\mathcal{O}(I)$ data along the edges of $G$. In total, the algorithm performs $\mathcal{O}(\text{scan}(N+I))$ I/Os.

Only few algorithms deal with problems on directed acyclic graphs. Also, the requirement that the vertices of the DAG be numbered in a manner consistent with a topological ordering of the graph is rather restrictive, but necessary because no $\mathcal{O}(\mathrm{sort}(N))$ I/O algorithm for topological sorting is known for general graphs. Still time-forward processing is an extremely powerful technique, as many abstract problems and problems on undirected graphs can be solved by transforming them into an evaluation problem of a directed acyclic graph. During the transformation, the numbering of the vertices is in our hands, so that a numbering of the vertices consistent with a topological ordering of the graph can often be obtained in an I/O-efficient manner.

### 5.2.4 List Ranking

Let $L$ be a linked list, i.e., a collection of nodes $x_1, \ldots, x_N$ such that each node $x_i$, except the tail of the list, stores a pointer $\mathrm{succ}(x_i)$ to its successor in list $L$, and there are no two nodes having the same successor. Given a pointer to the head of the list (i.e., the node which no other node in the list points to), the *list ranking* problem is that of computing for every node $x_i$ of list $L$, its distance from the tail of $L$, i.e., the number of edges on the path from $x_i$ to the tail of $L$.

Often we use the term "list-ranking" to denote the following generalization of the above problem: Given a function $\lambda : \{x_1, \ldots, x_N\} \to X$ assigning labels to the nodes of list $L$ and a multiplication $\otimes : X \times X \to X$ defined on $X$, compute a label $\phi(x_i)$ for each node $x_i$ of $L$ such that $\phi(x_{\sigma(1)}) = \lambda(x_{\sigma(1)})$ and $\phi(x_{\sigma(i)}) = \phi(x_{\sigma(i-1)}) \otimes \lambda(x_{\sigma(i)})$, for $1 < i \leq N$, where $\sigma : [1, N] \to [1, N]$ is the permutation so that $x_{\sigma(1)}$ is the head of $L$ and $\mathrm{succ}(x_{\sigma(i)}) = x_{\sigma(i+1)}$, for $1 \leq i < N$.

In internal memory, this problem can be solved by following pointers from the head of the list to its tail, keeping a value $\phi$ as the running product of the labels of the vertices visited so far, and defining for every vertex $x_i$, $\phi(x_i)$ as the value of $\phi$ at the time when vertex $x_i$ is visited. Unfortunately, this algorithm is not I/O-efficient: As we have no control over the physical order of the nodes of list $L$ on disk, an

adversary can easily arrange these nodes in a manner that forces the algorithm to perform one I/O per followed pointer. Hence, the algorithm takes $\Omega(N)$ I/Os in the worst case, while the lower bound for list-ranking shown in [43] is only $\Omega(\text{perm}(N))$. Next we sketch an $\mathcal{O}(\text{sort}(N))$ I/O algorithm for list ranking proposed in [43] which closes the gap.

We make the simplifying assumption that multiplication on $X$ is associative. If this is not the case, we first solve the classical list-ranking problem to determine the rank of every node, sort the nodes by their ranks and then scan the resulting list to compute the prefix product over the labels of the nodes in the sorted list.

Given that multiplication on $X$ is associative, the list-ranking algorithm of [43] applies the graph contraction paradigm of Section 5.2.2: First it finds an independent set $I$ of size $\Omega(N)$ in $L$. Then it removes the nodes in $I$ from $L$. For every node $x_i \in I$, a new label $\lambda'(\text{succ}(x_i)) = \lambda(x_i) \otimes \lambda(\text{succ}(x_i))$ of its successor is computed. For every node $x_i \notin I$, its new successor is defined as $\text{succ}'(x_i) = \text{succ}(x_i)$ if $\text{succ}(x_i) \notin I$, and $\text{succ}'(x_i) = \text{succ}(\text{succ}(x_i))$ if $\text{succ}(x_i) \in I$. For all nodes $x_i$ in the resulting list $L'$, $\phi(x_i) = \phi'(x_i)$, where $\phi'(x_i)$ is the label assigned to $x_i$ by applying the list-ranking algorithm to list $L'$ and labelling $\lambda'$. For every node $x_i \in I$, $\phi(x_i) = \phi'(x_j) \otimes \lambda(x_i)$, where $x_j \in L'$ is the node such that $\text{succ}(x_j) = x_i$. Given that it takes $\mathcal{O}(\text{sort}(N))$ I/Os to find and remove the independent set $I$, the whole list-ranking algorithm takes $\mathcal{I}(N) = \mathcal{O}(\text{sort}(N)) + \mathcal{I}(cN)$ I/Os, where $c < 1$. Thus, $\mathcal{I}(N) = \mathcal{O}(\text{sort}(N))$. In order to find the independent set $I$, Chiang et al. [43] apply a 3-coloring procedure for lists, which applies time-forward processing to "monotone" sublists of $L$ and takes $\mathcal{O}(\text{sort}(N))$ I/Os. The largest monochromatic set is chosen to be set $I$.

List ranking in itself is of very limited use. However, combined with the Euler tour technique described in the next section, it becomes a very powerful tool for solving graph problems that can be expressed as functions over a traversal of a spanning tree of the graph. An important application is the *rooting* of a tree $T$, which is the process of establishing parent-child relationships between adjacent vertices of $T$. Once these

relationships are established, the Euler tour technique and list-ranking can be used to compute preorder or postorder numberings of the vertices of $T$, or the sizes of the subtrees of $T$ rooted at these vertices. This information, when computed for a spanning tree of an arbitrary graph, can be used in simple graph partitioning schemes, or in algorithms based on the graph contraction technique described in Section 5.2.2.

### 5.2.5 The Euler Tour Technique

The *Euler tour* technique is a way to define a traversal of a tree $T = (V, E)$ so that every edge of the tree is traversed exactly twice, once in each direction. Such a traversal is useful, as it produces a linear list of vertices or edges which captures the structure of the tree. This allows the solution of problems on tree $T$ which can be expressed as functions over an Euler tour. Evaluating the latter functions is easier than working with the tree itself, as the Euler tour has a linear structure, while tree $T$ in general does not.

Formally, the goal is to build a linked list $L$ whose elements are the edges in the set $\{(v, w), (w, v) : \{v, w\} \in E\}$ and so that every edge shares one endpoint with its predecessor and the other endpoint with its successor in the list. In order to define an Euler tour, choose a circular order of the edges incident to each vertex of $T$. Let $\{v, w_1\}, \ldots, \{v, w_k\}$ be the edges incident to vertex $v$. Then let $\text{succ}((w_i, v)) = (v, w_{i+1})$, for $1 \leq i < k$, and $\text{succ}((w_k, v)) = (v, w_1)$. The result is a circular linked list of the edges in $T$. An Euler tour that starts at some vertex $r$ and returns to that vertex after traversing every edge of $T$ exactly twice is obtained by choosing an edge $(v, r)$ with $\text{succ}((v, r)) = (r, w)$, setting $\text{succ}((v, r)) =$ **null**, and choosing $(r, w)$ as the first edge of the traversal.

Applied to a tree $T$ rooted at a node $r$, this technique produces a depth-first traversal of $T$. After assigning appropriate labels to the elements of list $L$, the list-ranking algorithm from the previous section can be applied to compute the prefix sum over these labels along $L$. Depending on the choice of the labels, the outcome is a labelling of the vertices of $T$ with their distances from $r$, a preorder numbering of $T$,

a postorder numbering of $T$, etc. For example, by assigning a weight of 1 to every edge $(p(v), v)$ in $L$ and a weight of $-1$ to every edge $(v, p(v))$, where $p(v)$ denotes the parent of vertex $v$ in $T$, the prefix sum over these labels along $L$ assigns the distance $d(r, v)$ of vertex $v$ from the root $r$ in $T$ to every edge $(u, v)$ in $L$.

## 5.3 Primitive Operations

In this section, we define a number of simple graph labelling operations and graph transformations that can be carried out in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. We use these operations as the building blocks of our algorithms throughout this thesis.

Most of the operations described here require a total order on the vertex set $V$ and the edge set $E$ of the graph $G = (V, E)$. For the vertex set $V$, such a total order is defined by a unique numbering of the vertices in $G$ and the natural order defined on these numbers. For the edge set $E$, we assume that an edge $\{v, w\}$ is stored as the pair $(v, w)$, $v < w$. Then we define $(v, w) < (x, y)$, for edges $(v, w)$ and $(x, y)$ in $E$, if either $v < x$ or $v = x$ and $w < y$. We call this ordering the *lexicographical order* of $E$. Another ordering, called the *inverted lexicographical order* of $E$, defines $(v, w) < (x, y)$ if either $w < y$ or $w = y$ and $v < x$.

**Set difference.** Even though set difference is not a graph operation as such, we often apply it to the vertex and edge sets of a graph. To compute the difference $X \setminus Y$ of two sets $X$ and $Y$ drawn from a total order, we sort $X$ and $Y$ and scan the two resulting sorted lists to remove all elements in $Y$ from $X$. This can be done in a single scan, as the elements in $X \cap Y$ appear in the same order in $X$ and $Y$ after sorting the two lists.

**Duplicate removal.** Given a list $X = \langle x_1, \ldots, x_N \rangle$ with some entries potentially occurring more than once, the DuplicateRemoval operation computes a list $Y = \langle y_1, \ldots, y_q \rangle$ such that $\{x_1, \ldots, x_N\} = \{y_1, \ldots, y_q\}$, $y_j = x_{i_j}$, $i_1 < \cdots < i_q$, and $x_l \neq y_j$, for $1 \leq l < i_j$. That is, list $Y$ contains the first occurrences of all elements in $X$ in

sorted order. (Alternatively, list $Y$ may be required to store the last occurrences of all elements in $X$.) To compute $Y$, we scan $X$ and replace element $x_i$ with the pair $(x_i, i)$. Then we sort the resulting list $X'$ lexicographically. We scan this sorted list and remove all elements $(x, y)$, except the first one, for every $x$, from list $X'$. List $Y$ is now produced by sorting the elements $(x, y)$ in the computed sublist of $X'$ in inverted lexicographical order and scanning the resulting list to replace every pair $(x, y)$ with the single element $x$.

**Computing incident edges.** Given the vertex and edge sets $V$ and $E$ of a graph $G$ and a subset $V' \subseteq V$ of vertices, the IncidentEdges operation computes the set $E'$ of edges $\{v, w\} \in E$ such that $v \in V'$ and $w \in V \setminus V'$. To compute $E'$, we sort $V$ in increasing order, and $E$ in lexicographical order. Then we scan $V$ and $E$ and mark every edge in $E$ which has its first endpoint in $V$. We sort $E$ in inverted lexicographical order and scan $V$ and $E$ again to mark every edge in $E$ which has its second endpoint in $V$. In a final scan, we extract all edges from $E$ which have been marked exactly once. Changing the rule so that every edge which has been marked twice is chosen, we obtain operation InducedEdgeSet which computes the edge set of graph $G[V']$.

**Copying labels from edges to vertices.** Given a graph $G = (V, E)$ and a labelling $\lambda : E \to X$ of the edges in $E$, the SumEdgeLabels operation computes a labelling $\lambda' : V \to X$ of the vertices in $V$, where $\lambda'(v) = \bigoplus_{e \in E_v} \lambda(e)$, $E_v$ is the set of edges incident to $v$, and $\oplus$ is any given associative and commutative operator on $X$. To compute the labeling in $\mathcal{O}(\mathrm{sort}(N))$ I/Os, we sort $V$ in increasing order, and $E$ lexicographically. Then we scan $V$ and $E$ to compute labels $\lambda''(v) = \bigoplus_{e \in E'_v} \lambda(e)$, where $E'_v$ is the set of edges whose lower endpoint is $v$. Next we sort $E$ in inverted lexicographical order and scan $V$ and $E$ to compute labels $\lambda'(v) = \lambda''(v) \oplus \bigoplus_{e \in E''_v} \lambda(e)$, where $E''_v$ is the set of edges whose higher endpoint is $v$.

**Copying labels from vertices to edges.** Given a graph $G = (V, E)$ and a labelling $\lambda : V \to X$ of the vertices in $V$, the CopyVertexLabels operation computes a labelling $\lambda' : E \to X \times X$, where $\lambda'(\{v, w\}) = (\lambda(v), \lambda(w))$. This operation is easily carried out using a similar procedure as the one implementing operation SumEdgeLabels.

**Making adjacency lists from an edge set.** In some situations, it is necessary to transform a representation of a graph $G = (V, E)$ as the two sets $V$ and $E$ into a collection of adjacency lists of the vertices of $G$. We call this operation EdgesToAdjacencyLists. To perform this operation, we scan the edge set $E$ and append two entries $(v, w)$ and $(w, v)$ to a list $\mathcal{A}$, for every edge $\{v, w\} \in E$. Then we sort list $\mathcal{A}$ lexicographically. As a result, pairs $(v, w_1), \ldots, (v, w_k)$ are stored consecutively, for every vertex $v$ and all edges $\{v, w_1\}, \ldots, \{v, w_k\}$ incident to $v$. Thus, list $\mathcal{A}$ is the concatenation of adjacency lists $A(v)$, $v \in V$.

**Graph contraction.** Even though graph contraction has been introduced as a general paradigm in Section 5.2.2, we have not provided a general procedure to contract subgraphs of a graph $G = (V, E)$ I/O-efficiently. In order to perform operation ContractGraph, we assume that graph $G$ is represented as the sets $V$ and $E$, and a labelling of the vertices is given so that two vertices have the same label if and only if they are to be contracted into the same "supervertex" of the contracted graph.

To perform the contraction, we apply procedure CopyVertexLabels to replace every edge endpoint with the supervertex containing that endpoint. Then we apply operation DuplicateRemoval to the vertex and edge sets of $G$. Alternatively, we may wish to keep duplicate edges, in which case duplicates are removed only from $V$.

In some situations, the contraction information is given not as a vertex labelling, but as a separate set of vertex-supervertex pairs. This information can be transformed into a vertex labelling as follows: We sort the vertex set and the label set lexicographically. Now a single scan of these two lists suffices to label every vertex

with its containing supervertex, so that we obtain the required information for the procedure above.

# Chapter 6

# Greedy Algorithms

In this chapter, we describe a simple technique to obtain I/O-efficient algorithms for certain graph problems that can be solved using greedy algorithms in internal memory. Using this technique we obtain simple deterministic $\mathcal{O}(\text{sort}(|V|+|E|))$ I/O algorithms for finding a maximal matching or maximal independent set of an arbitrary graph.

Let us define precisely what we mean by "certain" graph problems. A *vertex labelling algorithm* is an algorithm $\mathcal{A}$ that computes a function $\lambda : V \to X$. We call $\mathcal{A}$ *single-pass* if it computes $\lambda$ by visiting every vertex $v \in V$ exactly once and assigns a label $\lambda(v)$ to $v$ during this visit. We call $\mathcal{A}$ *local* if $\lambda(v)$ can be computed in $\mathcal{O}(\text{sort}(k))$ I/Os from labels $\lambda(u_1), \ldots, \lambda(u_k)$, $u_1, \ldots, u_k \in \Gamma(v)$, which have been computed before visiting $v$. Finally, we call $\mathcal{A}$ *presortable* if there is an algorithm which takes $\mathcal{O}(\text{sort}(|V|+|E|))$ I/Os to establish an order so that $\mathcal{A}$ produces a correct result if it visits the vertices of $G$ in this order. We consider graph problems that can be solved using a presortable local single-pass vertex labelling algorithm.

**Theorem 6.1** *Every graph problem $\mathcal{P}$ which is solvable by a presortable local single-pass vertex labelling algorithm can be solved in $\mathcal{O}(\text{sort}(|V|+|E|))$ I/Os.*

*Proof.* We use Algorithm 6.1 to solve problem $\mathcal{P}$. Let $\mathcal{A}$ be a presortable local single-pass vertex labelling algorithm that solves problem $\mathcal{P}$. Since $\mathcal{A}$ is local and the

**Procedure** IOGREEDY

**Input:** A graph $G = (V, E)$ and a labelling problem $\mathcal{P}$ which can be solved using a presortable local single-pass vertex labelling algorithm $\mathcal{A}$.

**Output:** The labelling $\lambda : V \to X$ of the vertices of $G$ that would be computed by algorithm $\mathcal{A}$.

1: Establish an order $\prec$ of the vertices of graph $G = (V, E)$ so that algorithm $\mathcal{A}$ produces a correct result if it visits the vertices in $V$ in this order and sort the vertices of $G$ in this order. Number the vertices of $G$ in their order of appearance.
2: Replace every edge $\{v, w\} \in E$ by a directed edge $(v, w)$, $v \prec w$. Let $G'$ be the resulting DAG. Sort the edges of $G'$ by their source vertices.
3: **for** all vertices $v \in V$, in their order of appearance **do**
4: Let $\Gamma^-_{G'}(v) = \{u_1, \ldots, u_k\}$. Compute $\lambda(v)$ from $\lambda(u_1), \ldots, \lambda(u_k)$.
5: **end for**

**Algorithm 6.1**
A framework for I/O-efficient greedy algorithms.

ordering $\prec$ is chosen so that algorithm $\mathcal{A}$ solves problem $\mathcal{P}$ correctly if it processes the vertices of $G$ in this order, label $\lambda(v)$ can be computed from labels $\lambda(u_1), \ldots, \lambda(u_k)$, where $\{u_1, \ldots, u_k\} = \{u \in \Gamma_G(v) : u \prec v\} = \Gamma^-_{G'}(v)$, for every vertex $v \in V$. This establishes the correctness of Algorithm 6.1.

Line 1 of Algorithm 6.1 takes $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os because algorithm $\mathcal{A}$ is presortable. Line 2 can easily be carried out in $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os, once every edge $\{v, w\}$ has been "informed" about the numbers $\nu(v)$ and $\nu(w)$ assigned to its endpoints $v$ and $w$ in Line 1. To do the latter, we apply operation COPYVERTEXLABELS.

Given the vertices and edges of DAG $G'$ in sorted order, the loop in Lines 3–5 takes $\mathcal{O}(\mathrm{sort}(|E|))$ I/Os: Assuming that every vertex $v$ has labels $\lambda(u_1), \ldots, \lambda(u_k)$, where $\Gamma^-_{G'}(v) = \{u_1, \ldots, u_k\}$, at its disposal, labels $\lambda(v)$, $v \in V$, can be computed in $\mathcal{O}\left(\mathrm{sort}\left(\sum_{v \in V} |\Gamma^-_{G'}(v)|\right)\right) = \mathcal{O}(\mathrm{sort}(|E|))$ I/Os, by the locality of algorithm $\mathcal{A}$. We can use time-forward processing [11] to provide every vertex $v \in V$ with this information, which takes $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os, as only $\mathcal{O}(1)$ information is sent along every edge of $G'$. □

Next we apply Theorem 6.1 to obtain deterministic $\mathcal{O}(\text{sort}(|V| + |E|))$ I/O algorithms for finding a maximal independent set and a maximal matching in a graph $G$ and coloring a graph $G$ of degree $\Delta$ with at most $\Delta + 1$ colors.

## 6.1 Computing a Maximal Independent Set

The following simple algorithm computes a maximal independent set $S$ of a graph $G = (V, E)$ in internal memory: *Process the vertices in an arbitrary order. When a vertex $v \in V$ is visited, add it to $S$ if none of its neighbors is in $S$.* Translated into a labelling problem, the algorithm computes the characteristic function $\chi_S : V \to \{0, 1\}$ of $S$, where $\chi_S(v) = 1$ if $v \in S$, and $\chi_S(v) = 0$ if $v \notin S$. Also note that if $S$ is initially empty, then any neighbor $w$ of $v$ that is visited after $v$ cannot be in $S$ at the time when $v$ is visited, so that it is sufficient for $v$ to inspect all its neighbors that are visited before $v$ to decide whether or not $v$ should be added to $S$. With these modifications, we obtain a vertex-labelling algorithm that is presortable (since the order in which the vertices are visited is unimportant), local (since only previously visited neighbors of $v$ are inspected to decide whether $v$ has to be added to $S$, and a single scan of labels $\chi_S(u_1), \ldots, \chi_S(u_k)$ of $v$'s neighbors $u_1, \ldots, u_k$ is sufficient to decide whether at least one of them is in $S$), and single-pass. Showing the correctness of the algorithm is straightforward. Hence, we obtain the following result.

**Theorem 6.2** *Given an undirected graph $G = (V, E)$, a maximal independent set of $G$ can be found in $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os and linear space.*

## 6.2 Coloring Graphs of Bounded Degree

A *$k$-coloring* of a graph $G = (V, E)$ is a labelling $c : V \to \{1, \ldots, k\}$ of the vertices of $G$ so that for every edge $\{v, w\} \in E$, $c(v) \neq c(w)$.

The algorithm to compute a $(\Delta + 1)$-coloring of a graph $G$ of degree $\Delta$ is similar to the algorithm for computing a maximal independent set presented in the previous

section: *Process the vertices in an arbitrary order. When a vertex $v \in V$ is visited, assign a color $c(v) \in \{1, \ldots, \Delta + 1\}$ to vertex $v$ which has not been assigned to any neighbor of $v$.* The algorithm is presortable and single-pass for the same reasons as those that make the maximal independent set algorithm presortable and single-pass. The algorithm is local, as the color of $v$ can be determined as follows: Sort the colors $c(u_1), \ldots, c(u_k)$ of $v$'s in-neighbors $u_1, \ldots, u_k$. Then scan this list and assign the first color not in this list to $v$. This takes $\mathcal{O}(\text{sort}(k))$ I/Os.

**Lemma 6.1** *Given an undirected graph $G = (V, E)$ whose vertices have degree at most $\Delta$, a $(\Delta + 1)$-coloring of $G$ can be computed in $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os and linear space.*

*Proof.* Since the algorithm described above is presortable, local, and single-pass, the I/O-complexity follows from Theorem 6.1. The correctness of the algorithm is obvious if we can guarantee that for every vertex $v \in V$, there is a color $c(v) \in \{1, \ldots, \Delta + 1\}$ which has not been assigned to a neighbor of $v$. This, however, follows from the fact that $v$ has at most $\Delta$ neighbors. □

## 6.3 Computing a Maximal Matching

Finding a maximal matching is not quite as straightforward as computing a maximal independent set, as it is an edge-labelling problem. In particular, the algorithm is required to compute the characteristic function $\chi_{\mathcal{M}} : E \to \{0, 1\}$ of a maximal matching $\mathcal{M}$. We can easily transform the problem of finding a maximal matching of a graph $G = (V, E)$ into that of finding a maximal independent set of the graph

$$G' = (E, \{\{e, e'\} : \text{edges } e \text{ and } e' \text{ share an endpoint in } G\}).$$

However, graph $G'$ may have size $\Omega(|V|^2)$ even if $|E| = \mathcal{O}(|V|)$. (As an example, consider a wagon wheel, which is even planar.) Our goal is to construct a subgraph $G'' = (E, E'')$ of $G'$ with $E'' = \mathcal{O}(|E|)$ and describe a vertex-labelling problem

of $G''$ whose solution corresponds to a maximal matching of $G$. We begin with the description of a procedure to construct such a graph $G''$.

Given graph $G = (V, E)$, we number the edges in $E$ in their order of appearance in the edge list of $G$. Then we define $e_1 < e_2$ if $e_1$ has a smaller number than $e_2$ in this numbering. For every vertex $v \in V$ with incident edges $e_1 < \cdots < e_q$, we add edges $\{e_i, e_{i+1}\}$, $1 \leq i < q$, to $E''$. Every vertex $e \in G''$ has at most two in-edges and at most two out-edges, one in-edge and one out-edge per endpoint of edge $e \in G$. Hence, $|E''| = \mathcal{O}(|E|)$, as desired. The vertex set of graph $G''$ is already given. The computation of the edge set requires an application of procedure EdgesToAdjacencyLists, sorting each of the resulting adjacency lists, and scanning these lists to extract the edges of $G''$. It remains to describe a vertex-labelling problem of $G''$ whose solution corresponds to a maximal matching of $G$ and which can be solved by a presortable local single-pass algorithm.

Every vertex $e \in G''$ is contained in two paths $P_v$ and $P_w$ in $G''$, one per endpoint of edge $e = \{v, w\} \in G$. A subset $\mathcal{M} \subseteq E$ is a maximal matching of $G$ if and only if the characteristic function $\chi_{\mathcal{M}} : E \to \{0, 1\}$ of $\mathcal{M}$ has the following two properties:

(M1) For every path $P_v$, $v \in V$, $\sum_{e \in P_v} \chi_{\mathcal{M}}(e) \leq 1$.

(M2) For every edge $e = \{v, w\} \in G$, $\sum_{e' \in P_v \cup P_w} \chi_{\mathcal{M}}(e') \geq 1$.

Property (M1) expresses the fact that $\mathcal{M}$ is a matching, i.e., every vertex has at most one incident edge in $\mathcal{M}$. Property (M2) expresses the maximality of $\mathcal{M}$, i.e., every edge not in $\mathcal{M}$ has to share an endpoint with an edge in $\mathcal{M}$. We compute function $\chi_{\mathcal{M}}$ using Algorithm 6.2. This algorithm is presortable, as it uses the ordering of the edges in $E$ used to construct $G''$, it is obviously single-pass, and it is local by the way labels $\lambda(e)$ are computed. All that remains to show is that the algorithm is correct.

**Theorem 6.3** *Given an undirected graph $G = (V, E)$, a maximal matching $M \subseteq E$ of $G$ can be computed in $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os and linear space.*

*Proof.* In order to prove the correctness of Algorithm 6.2, we have to show that the labelling $\chi_{\mathcal{M}}(e)$ constructed by the algorithm has Properties (M1) and (M2). Since

**Procedure** MAXIMALMATCHING

**Input:** An undirected graph $G = (V, E)$.
**Output:** A maximal matching $\mathcal{M} \subseteq E$ of $G$.

1: Construct graph $G'' = (E, E'')$ as described in the text, and label every edge $\{e, e'\} \in E''$ with the name of the endpoint shared by edges $e$ and $e'$ in $G$.
2: Sort the vertex set $E$ of $G''$ by the relation "$<$" defined on the edges of $G$.
3: **for** every vertex $e \in E$, in their order of appearance **do**
4: Compute a label $\lambda(e) = (\chi_{\mathcal{M}}(e), \sigma(P_v, e), \sigma(P_w, e))$ of vertex $e$, where $\sigma(P, e) = \sum\{\chi_{\mathcal{M}}(e') : e' \in P \text{ and } e' \leq e\}$: Let $e = \{v, w\}$, and let $e_v$ and $e_w$ be the neighbors of $e$ in paths $P_v$ and $P_w$ so that $e_v < e$ and $e_w < e$. If $e_v$ does not exist, assume that $e_v$ is a dummy vertex with $\lambda(e_v) = (0, 0, 0)$. Make the same assumptions about $e_w$. Then let $\chi_{\mathcal{M}}(e) = 1$ if $\sigma(P_v, e_v) + \sigma(P_w, e_w) = 0$, and $\chi_{\mathcal{M}}(e) = 0$ otherwise. Let $\sigma(P_v, e) = \sigma(P_v, e_v) + \chi_{\mathcal{M}}(e)$ and $\sigma(P_w, e) = \sigma(P_w, e_w) + \chi_{\mathcal{M}}(e)$.
5: **end for**
6: Scan $E$ and extract label $\chi_{\mathcal{M}}(e)$ from label $\lambda(e)$, for every edge $e \in E$.

**Algorithm 6.2**
Computing a maximal matching.

the algorithm processes the vertices of $G''$ sorted by the "$<$" relation, it is easily verified that indeed $\sigma(P_v, e) = \sum\{\chi_{\mathcal{M}}(e') : e' \in P_v \text{ and } e' \leq e\}$ and $\sigma(P_w, e) = \sum\{\chi_{\mathcal{M}}(e') : e' \in P_w \text{ and } e' \leq e\}$. This immediately implies that labelling $\chi_{\mathcal{M}}$ has Property (M2) because the algorithm sets $\chi_{\mathcal{M}}(e) = 1$ unless $\sigma(P_v, e) + \sigma(P_w, e) \geq 1$. In both cases, Property (M2) holds. Property (M1) holds because $\chi_{\mathcal{M}}(e) = 1$ implies that $\sigma(P_v, e') = 0$, for all $e' \in P_v$ with $e' < e$, and $\sigma(P_v, e') = 1$, for all $e' \in P_v$ with $e \leq e'$. Hence, $\chi_{\mathcal{M}}(e') = 0$, for all $e' \in P_v \setminus \{e\}$. The same argument shows that $\chi_{\mathcal{M}}(e') = 0$, for all $e' \in P_w \setminus \{e\}$. The I/O-complexity follows from Theorem 6.1. □

Since a maximal independent set and a maximal matching of an arbitrary graph and a $(\Delta + 1)$-coloring of a graph of bounded degree can be computed in linear time in internal memory, we obtain the following corollary, using Lemma 2.1.

**Corollary 6.1** *The following problems can be solved in $\mathcal{O}(\text{perm}(|V| + |E|))$ I/Os and linear space: Computing a maximal independent set or maximal matching of a graph $G = (V, E)$ and computing a $(\Delta + 1)$-coloring of a graph $G$ of degree $\Delta$.*

# Chapter 7

# Outerplanar Graphs

In this chapter, we show that a number of fundamental graph problems can be solved in $\mathcal{O}(\text{scan}(N))$ I/Os on embedded outerplanar graphs, provided that the embedding is represented in an appropriate manner. The problems we consider are breadth-first search, depth-first search, single source shortest paths, triangulating an embedded outerplanar graph, and computing an $\varepsilon$-separator of size $\mathcal{O}(1/\varepsilon)$. Clearly, the I/O-complexities of our algorithms cannot be improved by more than a constant factor if no preprocessing is allowed.

We also present an $\mathcal{O}(\text{sort}(N))$ I/O algorithm to test whether a graph $G = (V, E)$ is outerplanar. The algorithm provides proof for its decision by providing an outerplanar embedding $\hat{G}$ of $G$ if $G$ is outerplanar, and by producing a subgraph of $G$ which is homeomorphic to $K_4$ or $K_{2,3}$ if $G$ is not outerplanar. Together with the above results, we thus obtain $\mathcal{O}(\text{sort}(N))$ I/O algorithms for the above problems on outerplanar graphs if no embedding is given. As there are linear-time internal memory algorithms for all of these problems on outerplanar graphs, Lemma 2.1 allows us to improve the I/O-complexities of our algorithms to $\mathcal{O}(\text{perm}(N))$. We prove matching lower bounds for BFS, DFS, SSSP, and embedding.

In our algorithms, we represent a graph $G = (V, E)$ as the two sets $V$ and $E$. An embedded outerplanar graph is represented as the set $V$ of vertices sorted clockwise along the outer boundary of the graph. Edges are represented as adjacency lists stored

with the vertices. Each adjacency list is sorted counterclockwise around the vertex, starting with the edge on the outer face of $G$ incident to $v$ and preceding $v$ in the clockwise traversal of the outer boundary of the graph. In the lower bound proof for outerplanar embedding, we use a slightly weaker representation of an embedding. In particular, the algorithm has to label every edge with its positions $n_v(e)$ and $n_w(e)$ counterclockwise around $v$ and $w$, respectively. This way we guarantee that the $\Omega(\text{perm}(N))$ I/O lower bound is not just a consequence of the requirement to arrange the vertices of $G$ in the right order.

In Section 7.1, we describe our algorithm for testing whether a given graph is outerplanar and computing an outerplanar embedding. In Section 7.2, we present an algorithm to triangulate a connected embedded outerplanar graph. In Section 7.3, we provide an algorithm for computing separators of embedded outerplanar graphs. In Section 7.4, we present algorithms for solving breadth-first search, depth-first search, and the single source shortest path problem on embedded outerplanar graphs. In Section 7.5, we prove lower bounds for embedding, DFS, BFS and SSSP on outerplanar graphs.

## 7.1 Outerplanarity Testing and Outerplanar Embedding

We begin our exposition of outerplanar graphs with a description of an I/O-efficient algorithm for outerplanarity testing and outerplanar embedding. The description is divided into three parts. In Section 7.1.1, we show how to find an outerplanar embedding of a biconnected outerplanar graph $G$. In Section 7.1.2, we show how to deal with the general case. In Section 7.1.3, we augment the algorithm so that it can test whether a given graph $G$ is outerplanar. If $G$ is outerplanar, the algorithm produces an outerplanar embedding of $G$. Otherwise, it outputs a subgraph of $G$ which is homeomorphic to $K_4$ or $K_{2,3}$. Graphs are assumed to be undirected in this section.

**Procedure** BiconnectedOuterplanarEmbedding

**Input:** A biconnected outerplanar graph $G$.
**Output:** An outerplanar embedding of $G$ represented by sorted adjacency lists.

1: Compute an open ear-decomposition $\mathcal{E} = (P_0, \ldots, P_k)$ of $G$.
2: Compute the cycle $C$ clockwise along the outer boundary of an outerplanar embedding $\hat{G}$ of $G$.
3: Compute for each vertex $v$ of $G$, its adjacency list sorted counterclockwise around $v$, starting with the predecessor of $v$ in $C$ and ending with the successor of $v$ in $C$.

**Algorithm 7.1**
Embedding a biconnected outerplanar graph.

### 7.1.1 Outerplanar Embedding of Biconnected Graphs

We use Algorithm 7.1 to compute an outerplanar embedding $\hat{G}$ of a biconnected outerplanar graph $G$. Next we describe the three steps of this algorithm in detail and prove the following lemma.

**Lemma 7.1** *Algorithm 7.1 takes $\mathcal{O}(\text{sort}(N))$ I/Os and linear space to compute an outerplanar embedding of a biconnected outerplanar graphs with $N$ vertices.*

**Step 1: Computing an open ear decomposition.** We apply the algorithm of [43] to compute an open ear-decomposition of $G$. This takes $\mathcal{O}(\text{sort}(N))$ I/Os and linear space. By Lemma 3.1, such an ear-decomposition of $G$ exists.

**Step 2: Computing the boundary cycle.** This step computes the boundary cycle $C$ of an outerplanar embedding $\hat{G}$ of $G$, i.e., the boundary cycle of the outer face of the embedding, which contains all vertices of $G$. The following lemma shows that $C$ is the only simple cycle containing all vertices of $G$, so that $C$ can be computed by computing *any* simple cycle with this property.

**Lemma 7.2** *Every biconnected outerplanar graph $G$ contains a unique simple cycle containing all vertices of $G$.*

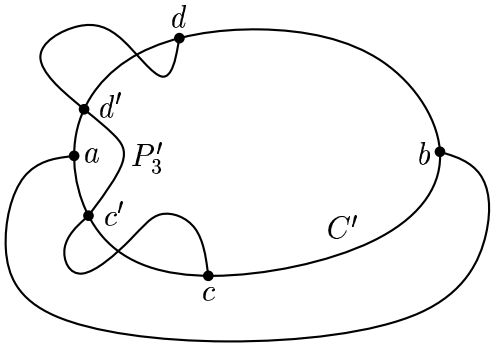



**Figure 7.1**
Proof of the uniqueness of the boundary cycle of a biconnected outerplanar graph.

*Proof.* The existence of cycle $C$ follows from the outerplanarity and biconnectivity of $G$. In particular, the boundary of the outer face of an outerplanar embedding $\hat{G}$ of $G$ is such a simple cycle.

So assume that there exists another simple cycle $C'$ containing all vertices of $G$. Let $a$ and $b$ be two vertices that are consecutive in $C$, but not in $C'$ (see Figure 7.1). Then cycle $C'$ can be broken into two internally vertex-disjoint paths $P_1$ and $P_2$ from $a$ to $b$, both of them containing at least one internal vertex. Let $c$ and $d$ be two internal vertices of $P_1$ and $P_2$, respectively. As $a$ and $b$ are connected by an edge in $C$, there must be a subpath $P_3$ of $C$ between $c$ and $d$ which contains neither $a$ nor $b$. Consider all vertices on $P_3$ that are also contained in $P_1$. Let $c'$ be such a vertex which is furthest away from $c$ along $P_3$. Analogously, we define $d'$ to be the vertex closest to $c$ which is shared by $P_3$ and $P_2$. Let $P_3'$ be the subpath of $P_3$ between $c'$ and $d'$. Let $H$ be the subgraph of $G$ defined as the union of cycle $C'$, edge $\{a, b\}$, and path $P_3'$. Cycle $C'$, edge $\{a, b\}$, and path $P_3'$ are internally vertex-disjoint. Thus, $H$ is homeomorphic to $K_4$, which contradicts the outerplanarity of $G$. □

Let $\mathcal{E} = \langle P_0, \ldots, P_k \rangle$ be the open ear decomposition computed in Step 1. Denote the endpoints of ear $P_i$, $1 \leq i \leq k$, by $\alpha_i$ and $\beta_i$. During the construction of $C$, we restrict our attention to the non-trivial ears $P_{i_0}, P_{i_1}, \ldots, P_{i_q}$, $0 = i_0 < i_1 < \cdots < i_q$, in $\mathcal{E}$, as a trivial ear does not contribute new vertices to the graph consisting of all

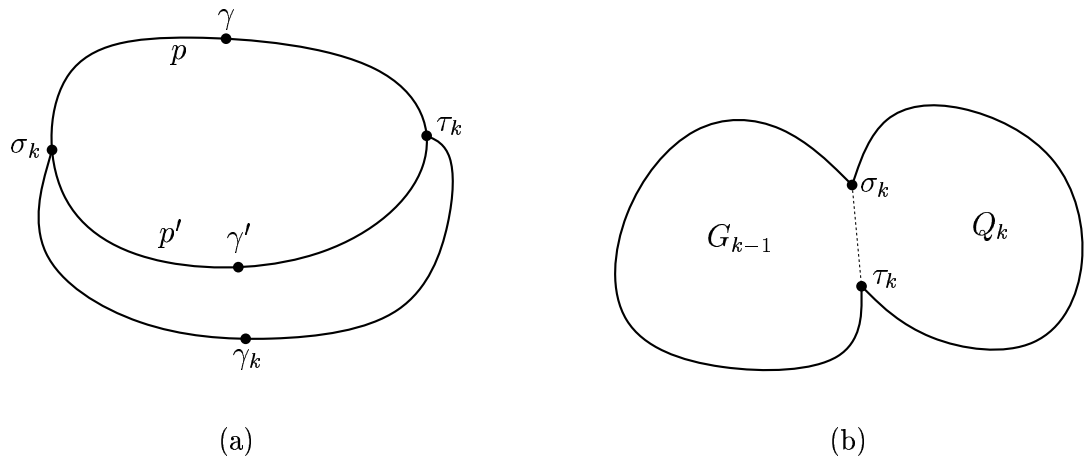



**Figure 7.2**
(a) The attachment vertices of $Q_k$ are adjacent in $G_{k-1}$. (b) This implies that $G_k$ is a cycle.

preceding ears. For the sake of simplicity, we write $Q_j = P_{i_j}$, $\sigma_j = \alpha_{i_j}$, and $\tau_j = \beta_{i_j}$, for $0 \leq i \leq q$. Let $G_1, \ldots, G_q$ be subgraphs of $G$ defined as follows: $G_1 = Q_0 \cup Q_1$. For $1 < i \leq q$, $G_i$ is obtained by removing edge $\{\sigma_i, \tau_i\}$ from $G_{i-1}$ and adding $Q_i$ to the resulting graph.

**Lemma 7.3** *Graphs $G_1, \ldots, G_q$ are simple cycles.*

*Proof.* If $i = 1$, $G_i$ is a simple cycle, as $Q_1$ is a simple path, and $Q_0$ is an edge connecting the two endpoints of $Q_1$. So assume that $G_i$ is a simple cycle for $1 \leq i < k \leq q$. We show that $G_k$ is a simple cycle.

The crucial observation is that the endpoints $\sigma_k$ and $\tau_k$ of $Q_k$ are adjacent in $G_{k-1}$. Assume the contrary. Then $G_{k-1}$ consists of two internally vertex-disjoint paths $P$ and $P'$ between $\sigma_k$ and $\tau_k$, each containing at least one internal vertex (see Figure 7.2a). Let $\gamma$ be an internal vertex of $P$, $\gamma'$ be an internal vertex of $P'$, and $\gamma_k$ be an internal vertex of $Q_k$. Vertex $\gamma_k$ exists because $Q_k$ is non-trivial and $k > 0$. Then paths $Q_k(\sigma_k, \gamma_k)$, $Q_k(\tau_k, \gamma_k)$, $P(\sigma_k, \gamma)$, $P(\tau_k, \gamma)$, $P'(\sigma_k, \gamma')$, and $P'(\tau_k, \gamma')$ are internally vertex disjoint, so that the graph $Q_k \cup G_{k-1}$, which is a subgraph of $G$, is homeomorphic to $K_{2,3}$. This contradicts the outerplanarity of $G$.

Given that vertices $\sigma_k$ and $\tau_k$ are adjacent in $G_{k-1}$, the removal of edge $\{\sigma_k, \tau_k\}$ from $G_{k-1}$ creates a simple path $G'$, so that $G'$ and $Q_k$ are two internally vertex-disjoint simple paths sharing their endpoints (see Figure 7.2b). Thus, $G_k = G' \cup Q_k$ is a simple cycle. □

By Lemmas 7.2 and 7.3, $G_q = C$. We have to show how to construct graph $G_q$. We obtain this graph by removing all trivial ears and all edges $\{\sigma_i, \tau_i\}$, $2 \leq i \leq q$, from $G$. Given an open-ear decomposition $\mathcal{E}$ of $G$, we scan $\mathcal{E}$ to create a list $L$ containing all trivial ears and edges $\{\sigma_i, \tau_i\}$, $2 \leq i \leq q$. Now we apply operation SetDifference to compute the edge set $E \setminus L$ of $C$. This takes $\mathcal{O}(\text{sort}(N))$ I/Os.

In order to complete this phase, the order of the vertices in $C$ along the outer boundary of $\hat{G}$ has to be established. To do this, we remove an arbitrary edge from $C$ to obtain a simple path $C'$. We compute the distance of each vertex in $C'$ from one endpoint of $C'$ using the Euler tour technique and list-ranking. These distances give the desired ordering of the vertices in $C$ along the outer boundary of $\hat{G}$. As argued in Sections 5.2.4 and 5.2.5, these two techniques take $\mathcal{O}(\text{sort}(N))$ I/Os and linear space.

**Step 3: Embedding the diagonals.** Let $C = (v_1, \ldots, v_N)$ be the boundary cycle computed in Step 2. In order to compute a complete embedding of $G$, the diagonals of $\hat{G}$ have to be embedded. (Diagonals are the edges removed from $G$ while computing $C$.) Assuming that the order of vertices in $C$ computed in the previous step is clockwise around the outer boundary of $\hat{G}$, such an embedding of the diagonals can be computed by sorting the adjacency lists $A(v_i)$ of vertices $v_i$, $1 \leq i \leq N$, counterclockwise around $v_i$, starting with $v_{i-1}$ and ending with $v_{i+1}$, where $v_{N+1} = v_1$ and $v_0 = v_N$.

We denote $\nu(v_i) = i$ as the *index* of vertex $v_i$, $1 \leq i \leq N$. First we apply procedure CopyVertexLabels to inform every edge about the indices of its endpoints. Then we use procedure EdgesToAdjacencyLists to transform the edge set into a set of adjacency lists for the vertices of $G$. The sorting step of procedure EdgesToAdjacencyLists can easily be modified to produce the adjacency lists

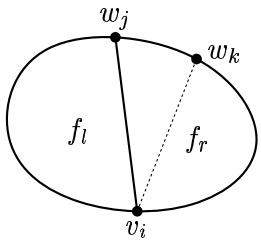



**Figure 7.3**
Proof of Lemma 7.4.

sorted by decreasing indices. Let $A(v_i) = \langle w_1, \ldots, w_t \rangle$ be the sorted adjacency list of vertex $v_i$. We scan the concatenation of lists $A(v_1), \ldots, A(v_N)$ to identify the vertex $w_j = v_{i-1}$ in each list $A(v_i)$ and rearrange the vertices in $A(v_i)$ to produce the list $\langle w_j, \ldots, w_t, w_1, \ldots, w_{j-1} \rangle$. The whole construction takes $\mathcal{O}(\text{sort}(N))$ I/Os. Let $A(v_i) = \langle w'_1, \ldots, w'_t \rangle$ be the final adjacency list of vertex $v_i$. We define $\nu_i(w'_j) = j$. The following lemma proves that the counterclockwise order of the vertices in $A(v_i)$ is computed correctly by sorting $A(v_i)$ as just described.

**Lemma 7.4** *Let $w_j$ and $w_k$ be two vertices in $A(v_i)$, $1 \leq i \leq N$, with $\nu_i(w_j) < \nu_i(w_k)$. Then $v_{i-1}$, $w_j$, $w_k$, and $v_{i+1}$ appear in this order counterclockwise around $v_i$ in an outerplanar embedding $\hat{G}$ of $G$.*

*Proof.* Consider the planar subdivision $D$ induced by $C$ and edge $\{v_i, w_j\}$ (see Figure 7.3). Subdivision $D$ has three faces: the outer face, a face $f_l$ bounded by the path from $v_i$ to $w_j$ clockwise along $C$ and edge $\{v_i, w_j\}$, and a face $f_r$ bounded by the path from $v_i$ to $w_j$ counterclockwise along $C$ and edge $\{v_i, w_j\}$. If $w_k$ appears between $v_{i-1}$ and $w_j$ in counterclockwise order around $v_i$, then $w_k$ is on the boundary of $f_r$, as edge $\{v_i, w_k\}$ and the boundary cycle of $f_r$ cannot intersect, except in vertices $v_i$ and $w_k$. Depending on where vertex $x$ with $\nu(x) = N$ appears along $C$ w.r.t. $v_{i-1}$,

$v_i$, $w_j$, and $w_k$, one of the following is true:

$$\nu(v_{i-1}) < \nu(v_i) < \nu(w_j) < \nu(w_k)$$
$$\nu(v_i) < \nu(w_j) < \nu(w_k) < \nu(v_{i-1})$$
$$\nu(w_j) < \nu(w_k) < \nu(v_{i-1}) < \nu(v_i)$$
$$\nu(w_k) < \nu(v_{i-1}) < \nu(v_i) < \nu(w_j)$$

It is easy to verify that in all four cases, $\nu_i(w_k) < \nu_i(w_j)$, contradicting the assumption made in the lemma. Thus, the lemma holds. □

### 7.1.2 Outerplanar Embedding—The General Case

If $G$ is not connected, the connected components $G_1, \ldots, G_k$ of $G$ can be embedded independently. An outerplanar embedding $\hat{G}_i$ of any component $G_i$, $1 \leq i \leq k$, can be obtained from outerplanar embeddings of its bicomps $\mathcal{B}_{i,1}, \ldots, \mathcal{B}_{i,l_i}$. In particular, the order of the vertices around every vertex $v$ that is contained in only one bicomp $\mathcal{B}_{i,j}$ is fully determined by the embedding of $\mathcal{B}_{i,j}$. For a cutpoint $v$ contained in bicomps $\mathcal{B}_{i,j_1}, \ldots, \mathcal{B}_{i,j_q}$, a valid ordering of the vertices around $v$ can be obtained by concatenating the sorted adjacency lists $A_1(v), \ldots, A_q(v)$ of $v$ in $\hat{\mathcal{B}}_{i,j_1}, \ldots, \hat{\mathcal{B}}_{i,j_q}$.

Similar to the case where $G$ is biconnected, our algorithm computes a list $\mathcal{A}$ which is the concatenation of sorted adjacency lists $A(v)$, $v \in G$. List $\mathcal{A}$ is the concatenation of lists $\mathcal{A}_1, \ldots, \mathcal{A}_k$, one for each connected component $G_i$, $1 \leq i \leq k$, of $G$. For the algorithms in Section 7.2 through 7.4 to run in $\mathcal{O}(\text{scan}(N))$ I/Os, the adjacency lists $A(v)$ in each list $\mathcal{A}_i$ need to be arranged in an appropriate order. For the biconnected case, this order is provided by the order of the vertices along the outer boundary of $G_i$. In the general case, this arrangement has to be done more carefully. In particular, we fix a clockwise traversal of the outer boundary of $G_i$, for every connected component $G_i$ of $G$. Then we sort every adjacency list $A(v)$, $v \in G_i$ counterclockwise around $v$ so that the first entry in $A(v)$ is the vertex preceding the first appearance of $v$ in this traversal. We arrange the adjacency lists $A(v_1), \ldots, A(v_{N_i})$ of the vertices in $G_i$ in the same order as the first appearances of vertices $v_1, \ldots, v_{N_i}$

**Procedure** OUTERPLANAREMBEDDING

**Input:** An outerplanar graph $G$.
**Output:** An outerplanar embedding of $G$ represented by sorted adjacency lists.

1: Compute the connected and biconnected components of $G$. Denote the connected components of $G$ by $G_1, \ldots, G_k$. The bicomps of component $G_i$ are denoted by $\mathcal{B}_{i,1}, \ldots, \mathcal{B}_{i,l_i}$.
2: Apply Algorithm 7.1 to all bicomps $\mathcal{B}_{i,j}$ of $G$ to compute lists $\mathcal{A}_{i,j}$ representing outerplanar embeddings of these bicomps.
3: **for** $i = 1, \ldots, k$ **do**
4: Apply Algorithm 7.3 to compute a list $C_i = \langle v_1, \ldots, v_{N_i} \rangle$ of the vertices of $G_i$, sorted by their first appearances in a clockwise traversal of the outer boundary of $G_i$.
5: Apply Algorithm 7.4 to compute a list $\mathcal{A}_i = A(v_1) \circ \cdots \circ A(v_{N_i})$, where $A(v_j)$ is the adjacency list of vertex $v_j$ sorted counterclockwise around $v_j$, starting with the predecessor of the first appearance of $v_j$ in a clockwise traversal of the outer boundary of $G_i$ and ending with its successor.
6: **end for**
7: $\mathcal{A} \leftarrow \mathcal{A}_1 \circ \cdots \circ \mathcal{A}_k$
8: $C \leftarrow C_1 \circ \cdots \circ C_k$
9: Let $C = \langle v_1, \ldots, v_N \rangle$. Then let $\nu(v_j) = j$ be the *index* of vertex $v_j$.

**Algorithm 7.2**
Embedding an outerplanar graph.

in the chosen traversal of the boundary of $G_i$. Algorithm 7.2 provides the details of computing list $\mathcal{A}$.

**Lemma 7.5** *Algorithm 7.2 takes $\mathcal{O}(\text{sort}(N))$ I/Os and linear space to compute an outerplanar embedding of an outerplanar graph with $N$ vertices.*

*Proof.* By Lemma 7.1, Algorithm 7.1 produces correct embeddings of all bicomps of $G$. Under this condition, Lemmas 7.6 and 7.7 show that Algorithms 7.3 and 7.4 compute correct representations $\mathcal{A}_1, \ldots, \mathcal{A}_k$ of outerplanar embeddings of the connected components $G_1, \ldots, G_k$ of $G$. Thus, Algorithm 7.2 correctly produces a list $\mathcal{A}$ representing an outerplanar embedding of $G$.

The computation of the connected and biconnected components of $G$ in Line 1 of the algorithm takes $\mathcal{O}(\text{sort}(N))$ I/Os using algorithms of [43]. By Lemma 7.1, Line 2

**Procedure** ComputeBoundary

**Input:** A connected outerplanar graph $G = (V, E)$ and lists $\mathcal{A}_1, \ldots, \mathcal{A}_k$ representing outerplanar embeddings of its bicomps $\mathcal{B}_1, \ldots, \mathcal{B}_k$.

**Output:** A list $C$ of the vertices of $G$, sorted by their first appearances in a clockwise traversal of the outer boundary of $G$. The first vertex in $C$ is not a cutpoint.

1: **if** $G$ consists of a single edge $\{v, w\}$ **then**
2: $C \leftarrow \langle v, w \rangle$
3: **else**
4: Compute a graph $G'$ by removing the diagonals of bicomps $\mathcal{B}_1, \ldots, \mathcal{B}_k$ from $G$.
5: **for** every cutpoint $v \in G$ **do**
6: Let $\mathcal{B}'_1, \ldots, \mathcal{B}'_q$ be the bicomps of $G$ containing $v$.
7: Let $u_i$ and $w_i$ be the predecessor and successor of $v$ clockwise along the outer boundary of bicomp $\mathcal{B}'_i$, for $1 \leq i \leq q$.
8: Make $q$ copies $v_1, \ldots, v_q$ of vertex $v$.
9: **for** $j = 1, \ldots, q$ **do**
10: Make $v_j$ adjacent to vertices $u_j$ and $w_{(j \bmod k)+1}$
11: **end for**
12: **end for**
13: Remove an edge $\{x, y\}$ from the resulting cycle $C'$ to transform it into a path $P$, where $x$ is not a cutpoint of $G$, and $y$ precedes $x$ in $C'$.
14: Compute distances $\text{dist}_P(x, z)$, $z \in P$.
15: Sort the vertices of $P$ by their distances and remove all but the first appearance of each vertex from $P$.
16: Let $C = \langle v_1, \ldots, v_N \rangle$ be the resulting list.
17: **end if**

**Algorithm 7.3**
Computing the list of vertices along the boundary of a connected outerplanar graph $G$.

takes $\mathcal{O}(\text{sort}(|\mathcal{B}_{i,j}|))$ I/Os per bicomp $\mathcal{B}_{i,j}$, $\mathcal{O}(\text{sort}(N))$ I/Os in total. By Lemmas 7.6 and 7.7, the $i$-th iteration of the loop in Lines 3–6 takes $\mathcal{O}(\text{sort}(|G_i|))$ I/Os. Hence, the whole loop takes $\mathcal{O}(\text{sort}(N))$ I/Os. Lines 7 and 8 do not require any computation, as lists $\mathcal{A}$ and $C$ can be created on the fly while producing lists $\mathcal{A}_1, \ldots, \mathcal{A}_k$ and $C_1, \ldots, C_k$ in the loop in Lines 3–6. Line 9 can be implemented in a single scan over list $C$. □

**Lemma 7.6** *Given a connected outerplanar graph $G = (V, E)$ with $N$ vertices, Algorithm 7.3 takes $\mathcal{O}(\text{sort}(N))$ I/Os and linear space to compute the list $C$ of the vertices of $G$, sorted by their first appearances in a clockwise traversal of the outer boundary of an outerplanar embedding of $G$.*

*Proof.* The correctness of Algorithm 7.3 is easily verified. If graph $G$ contains only a single edge $\{v, w\}$, $C = \langle v, w\rangle$ is a correct representation of its outer boundary. Otherwise, Lines 4–12 compute a cycle $C'$ so that all vertices of $G$ appear in the same order in $C'$ as visited by a clockwise traversal of an outerplanar embedding of $G$, including multiplicities (see Figure 7.4). Then Lines 13–16 sort the vertices of $G$ according to the order of their first appearances in $C'$ and hence along the outer boundary of $G$. Vertex $x$, which is chosen not to be a cutpoint, is the first vertex in $C$.

If graph $G$ consists of a single edge, Algorithm 7.3 clearly takes $\mathcal{O}(\text{sort}(N))$ I/Os. So assume that $G$ contains at least two edges. Line 4 of the algorithm can be realized by scanning lists $\mathcal{A}_1, \ldots, \mathcal{A}_k$ and removing all but the first and last entries from every adjacency list $A_i(v)$. Let $\mathcal{A}'$ be the resulting concatenation of adjacency lists. Assume that every vertex $w \in A_i(v)$ is represented as an ordered pair $(v, w)$. We sort the entries $(v, w) \in \mathcal{A}'$ by their first components using a stable sorting algorithm.[1] As a result, the edges in $G'$ incident to every vertex are stored consecutively, sorted by the bicomps containing them. Now a single scan of this list is sufficient to identify the cutpoints of $G$ and perform the loop in Line 5–12. Given that the cutpoints of $G$ have been identified, a single scan of the vertex list of $G$ suffices to find a vertex $x$ which is not a cutpoint. A scan of the edge list of $C'$ is sufficient to find and remove the edge $\{x, y\}$ preceding $x$ in a clockwise traversal of the outer boundary of $G$. Computing the distances of all vertices in the resulting path $P$ from $x$ takes $\mathcal{O}(\text{sort}(N))$ I/Os using the Euler tour technique and list-ranking. Line 15 now requires the sorting of the vertices of $P$ by their distances from $x$ and the application of operation DUPLICATEREMOVAL

[1] Any sorting algorithm can be made stable by including the original position of every element as a secondary key of the sort.

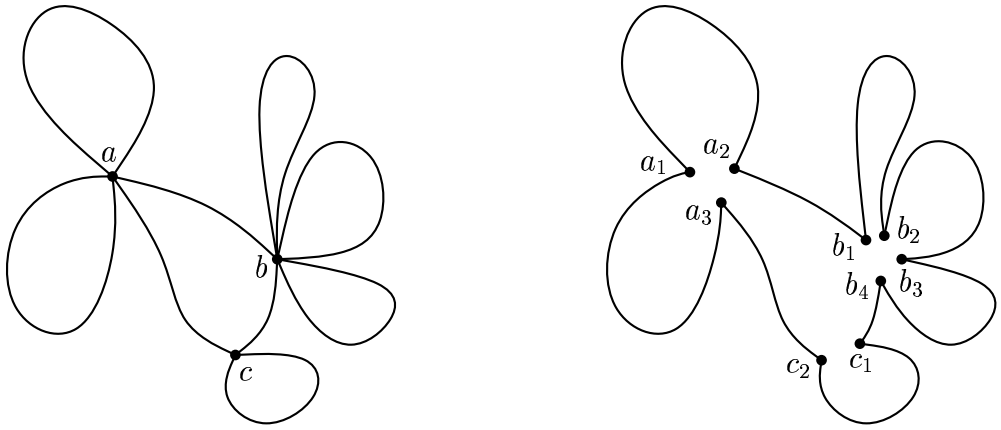



**Figure 7.4**
Extracting the list of vertices along the outer boundary of a connected outerplanar graph.

to remove all but the first appearance of every vertex. As all operations used here take $\mathcal{O}(\text{sort}(N))$ I/Os, the whole algorithm takes $\mathcal{O}(\text{sort}(N))$ I/Os. □

**Lemma 7.7** *Algorithm 7.4 takes $\mathcal{O}(\text{sort}(N))$ I/Os to compute a list $\mathcal{A}$ representing an outerplanar embedding of a connected outerplanar graph $G$ with $N$ vertices.*

*Proof.* To prove the correctness of the algorithm, it suffices to show that it computes the adjacency lists of all vertices correctly because Line 12 computes list $\mathcal{A}$ from lists $C$ and $A(v_1), \ldots, A(v_N)$ according to its definition. For a vertex $v$ which is not a cutpoint, its adjacency list $A(v)$ in $G$ is the same as its adjacency list $A_i(v)$ in the bicomp $\mathcal{B}_i$ of $G$ containing $v$. Thus, the algorithm correctly copies this adjacency list in Line 9. If vertex $v$ is a cutpoint, we have to show that the vertices in $A(v)$ are sorted counterclockwise around $v$, starting with the predecessor $u_x$ of the first appearance of $v$ in a clockwise traversal of the outer boundary of $G$. To do this, it suffices to show that $u_x = u_{\sigma(1)}$, and vertices $u_{\sigma(1)}, u_{\sigma(q)}, \ldots, u_{\sigma(2)}$ appear in this order counterclockwise around $v$.

As vertex $v_1$ is not a cutpoint, the vertex $u_x$ preceding the first appearance of $v$ in a clockwise traversal of the outer boundary of $G$ precedes $v$ in $C$. Let $\mathcal{B}'_x$ be the bicomp containing $u_x$. Then any vertex in any other bicomp $\mathcal{B}'_i$, $i \neq x$, can only be visited

**Procedure** ComputeAdjacencyLists

**Input:** A connected outerplanar graph $G = (V, E)$, lists $\mathcal{A}_1, \ldots, \mathcal{A}_k$ representing outerplanar embeddings of its bicomps $\mathcal{B}_1, \ldots, \mathcal{B}_k$, and a list $C = \langle v_1, \ldots, v_N \rangle$ of the vertices of $G$, sorted by their first appearances in a clockwise traversal of the outer boundary of $G$. Vertex $v_1$ is not a cutpoint.

**Output:** A list $\mathcal{A}$ representing an outerplanar embedding of $G$.

1: **for** $v \in V$ **do**
2: **if** $v$ is a cutpoint **then**
3: Let $\mathcal{B}'_1, \ldots, \mathcal{B}'_q$ be the bicomps containing $v$, and let $A_1(v), \ldots, A_q(v)$ be the adjacency lists of $v$ contained in lists $\mathcal{A}'_1, \ldots, \mathcal{A}'_q$.
4: Let $u_1, \ldots, u_q$ be the first entries of adjacency lists $A_1(v), \ldots, A_q(v)$.
5: Let $\sigma : [1, q] \to [1, q]$ be a permutation so that vertices $u_{\sigma(1)}, \ldots, u_{\sigma(q)}$ appear in this order in $C$.
6: $A(v) \leftarrow A_{\sigma(1)}(v) \circ A_{\sigma(q)}(v) \circ A_{\sigma(q-1)}(v) \circ \cdots \circ A_{\sigma(2)}(v)$
7: **else**
8: Let $\mathcal{B}_i$ be the only bicomp containing $v$, and let $A_i(v)$ be the adjacency list of $v$ contained in $\mathcal{A}_i$.
9: $A(v) \leftarrow A_i(v)$
10: **end if**
11: **end for**
12: $\mathcal{A} \leftarrow A(v_1) \circ \cdots \circ A(v_N)$

**Algorithm 7.4**
Computing an outerplanar embedding of a connected outerplanar graph $G$.

after visiting $v$ for the first time because $v$ is a cutpoint. Hence, they succeed $v$ and thus $u_x$ in $C$. Since the vertices in $C$ are stored clockwise along the outer boundary of $G$, vertices $u'_{\sigma(1)}, u'_{\sigma(q)}, \ldots, u'_{\sigma(2)}$ appear in this order counterclockwise around $v$.

It remains to provide the details of Algorithm 7.4 and analyze its I/O-complexity. First we apply a procedure similar to procedure CopyVertexLabels to inform every entry $(v, w)$ in an adjacency list $A_i(v)$ about the position of vertex $w$ in list $C$. This takes $\mathcal{O}(\text{sort}(N))$ I/Os. We scan the concatenation of all adjacency lists $A_i(v)$, for all bicomps $\mathcal{B}_1, \ldots, \mathcal{B}_k$ and all vertices $v \in V$ to label every entry $(v, w) \in A_i(v)$ with the position of the first vertex $u_i \in A_i(v)$ in $C$. Then we use a stable sort to arrange these adjacency lists so that for every vertex $v$ contained in bicomps $\mathcal{B}'_1, \ldots, \mathcal{B}'_q$,

adjacency lists $A'_1(v), \ldots, A'_q(v)$ are stored consecutively, arranged in reverse order by the position of their first entries in $C$. Now a single scan of the resulting list is sufficient to bring for every cutpoint $v$, adjacency list $A'_{\sigma(1)}(v)$ to the front of adjacency list $A(v)$. □

We conclude this subsection with a number of definitions and two observations which are used in the correctness proofs of algorithms presented in Sections 7.2 through 7.4.

**Observation 7.1** *For every adjacency list $A(v) = \langle w_1, \ldots, w_k \rangle$, $\nu(v) > 1$, there exists an index $1 \leq j \leq k$ such that $\nu(w_{j+1}) > \nu(w_{j+2}) > \cdots > \nu(w_k) > \nu(v) > \nu(w_1) > \cdots > \nu(w_j)$. If $\nu(v) = 1$, then $\nu(w_1) > \cdots > \nu(w_k) > \nu(v)$.*

We call a path $P = (v_0, \ldots, v_p)$ *monotone* if $\nu(v_0) < \cdots < \nu(v_p)$. We say that in the computed embedding of $G$, a monotone path $P = (s = v_0, \ldots, v_p)$ is *to the left* of another monotone path $P' = (s = v'_0, \ldots, v'_{p'})$ with the same source $s$ if $P$ and $P'$ share vertices $s = w_0, \ldots, w_t$, i.e., $w_k = v_{i_k} = v'_{j_k}$, for some indices $i_k$ and $j_k$ and $0 \leq k \leq t$, and edges $\{w_k, v_{i_k-1}\}$, $\{w_k, v_{i_k+1}\}$ and $\{w_k, v'_{j_k+1}\}$ appear in this order clockwise around $w_k$, for $0 \leq k < t$. This definition is somewhat misleading, as vertex $v_{-1}$ is not defined. But we apply results based on this definition only to monotone paths starting at the vertex $r$, $\nu(r) = 1$, of a connected embedded outerplanar graph $G$. In this case, we imagine $v_{-1}$ to be an additional vertex embedded in the outer face of the given outerplanar embedding of $G$ and connected only to $r$ through a single edge $\{v_{-1}, r\}$.

A *lexicographical ordering* "$\prec$" of the monotone paths with the same source is defined as follows: Let $P = (v_0, \ldots, v_p)$ and $P' = (v'_0, \ldots, v'_{p'})$ be two such paths. Then there exists an index $j > 0$ such that $v_k = v'_k$, for $0 \leq k < j$ and $v_j \neq v'_j$. We say that $P \prec P'$ if and only if $\nu(v_j) < \nu(v'_j)$.

**Lemma 7.8**

*(i) Let $P$ be a monotone path from $s$ to some vertex $v$. Then there exists no monotone path from $s$ to a vertex $w$, $\nu(w) > \nu(v)$, which is to the left of $P$.*

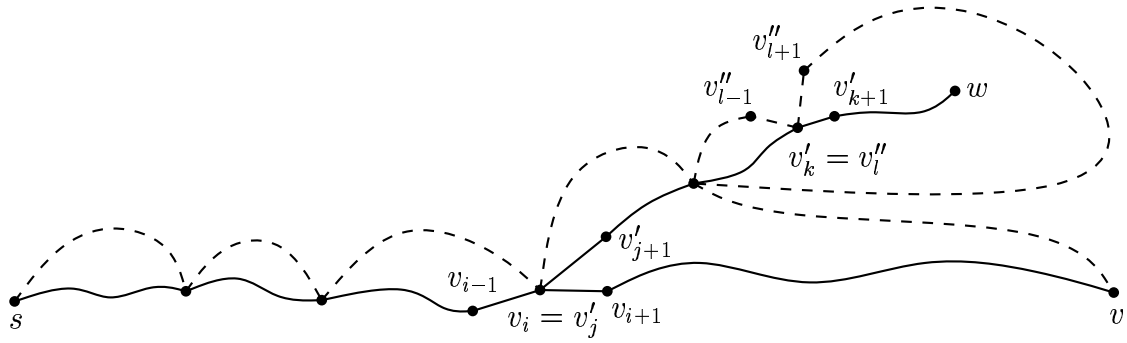



**Figure 7.5**
Proof of Lemma 7.8.

*(ii) Let $P$ and $P'$ be two monotone paths with source $s$, $\nu(s) = 1$. If $P \prec P'$, then $P'$ is not to the left of $P$.*

*Proof.* (i) Let $P = (s = v_0, \ldots, v_p = v)$ be a monotone path from $s$ to a vertex $v$, $P' = (s = v'_0, \ldots, v'_{p'} = w)$ be a monotone path from $s$ to a vertex $w$, $\nu(w) > \nu(v)$, and $v_i = v'_j$ be the last vertex shared by $P$ and $P'$ (see Figure 7.5). Assume that $P'$ is to the left of $P$. Then vertices $v_{i-1}$, $v_{i+1}$, and $v'_{j+1}$ appear in this order counterclockwise around $v_i$. As the vertices of $G$ are numbered clockwise along the outer boundary, the path $P''$ from $s$ to $v$ clockwise along the outer boundary includes $v_i$, but not $w$. Let $v'_k = v''_l$ be the last vertex along $P'$ shared by $P'$ and $P''$. As $P''$ is part of the outer boundary of $G$, vertices $v''_{l-1}$, $v'_{k+1}$, and $v''_{l+1}$ appear in this order counterclockwise around $v'_k$. But this implies that paths $P(v_i, v)$ and $P''(v_i, v)$ define a collection of closed Jordan curves one of which encloses $w$. This contradicts the assumption that the orders of the edges around the vertices of $G$ describe an outerplanar embedding of $G$. Hence, $P'$ is not to the left of $P$.

(ii) Let $P = (s = v_0, \ldots, v_p)$ and $P' = (s = v'_0, \ldots, v'_{p'})$ be two monotone paths with source $s$ so that $P \prec P'$. Assume that $P'$ is to the left of $P$. Let $i$ be the index so that $v_j = v'_j$, for $0 \leq j < i$, and $v_i \neq v'_i$. Then $\nu(v_i) < \nu(v'_i)$ and path $P'(s, v'_i)$ is to the left of path $P(s, v_i)$. This contradicts (i). □

### 7.1.3 Outerplanarity Testing

In this section, we augment the embedding algorithm of the previous section so that it decides whether a given graph $G = (V, E)$ is outerplanar. First the algorithm tests whether $|E| \leq 2|V| - 3$. If not, $G$ cannot be outerplanar. As a graph is outerplanar if and only if its bicomps are outerplanar, we only have to augment Algorithm 7.1, which deals with the bicomps of $G$. If this algorithm produces an outerplanar embedding for every bicomp of $G$, this is proof that $G$ is outerplanar. Otherwise, the algorithm fails to produce an outerplanar embedding for at least one of the bicomps of $G$. Let $\mathcal{B}$ be such a bicomp.

The algorithm can fail in two ways. It may not be able to compute the boundary cycle $C$, or it computes the boundary cycle $C$ and then produces an intersection between two edges when embedding the diagonals of $\mathcal{B}$. We discuss both cases in detail.

Given an open ear-decomposition $\mathcal{E} = \langle P_0, \ldots, P_k \rangle$ of $\mathcal{B}$, the algorithm tries to compute the boundary cycle $C$ by producing a sequence of cycles $G_0, \ldots, G_q$, where $G_{i+1}$ is obtained from $G_i$ by replacing edge $\{\sigma_{i+1}, \tau_{i+1}\}$ in $G_i$ by the non-trivial ear $Q_{i+1}$. If $G_i$ contains edge $\{\sigma_{i+1}, \tau_{i+1}\}$, for all $0 \leq i < q$, the algorithm successfully computes $C$. The only way this construction can fail is that there is a non-trivial ear $Q_{i+1}$ such that $G_i$ does not contain edge $\{\sigma_{i+1}, \tau_{i+1}\}$. As shown in the proof of Lemma 7.3, $G_i \cup Q_{i+1}$ is homeomorphic to $K_{2,3}$ in this case. Thus, the algorithm outputs $G_i \cup Q_{i+1}$ as proof that $G$ is not outerplanar.

Given the boundary cycle $C$, all edges of $\mathcal{B}$ which are not in $C$ are diagonals of $\mathcal{B}$. The algorithm computes list $\mathcal{A}$ as described in Section 7.1.1 and uses $\mathcal{A}$ and a stack $S$ to test for intersecting diagonals. The details are provided in Algorithm 7.5.

**Lemma 7.9** *Given cycle $C$ and list $\mathcal{A}$ as computed by Algorithm 7.1, bicomp $\mathcal{B}$ is outerplanar if and only if Algorithm 7.5 confirms this.*

*Proof.* If graph $G$ is not outerplanar, but cycle $C$ exists, two diagonals must intersect. Let $\{a, b\}$ and $\{c, d\}$ be two such edges. Since no two edges sharing an endpoint

**Procedure** TESTDIAGONALS

**Input:** The list $\mathcal{A}$ as computed for bicomp $\mathcal{B}$ by Algorithm 7.1.
**Output:** A decision whether $\mathcal{B}$ is outerplanar, along with a proof for the decision, either in the form of a subgraph which is homeomorphic to $K_4$, or in the form of an outerplanar embedding of $\mathcal{B}$.

1: Initialize stack $S$ to be empty.
2: **for** each entry $(v, w) \in \mathcal{A}$ **do**
3: **if** $\nu(w) > \nu(v)$ **then**
4: PUSH$(S, \{v, w\})$
5: **else**
6: $\{a, b\} \leftarrow$ POP$(S)$
7: **if** $\{a, b\} \neq \{v, w\}$ **then**
8: Report the graph consisting of cycle $C$ augmented with edges $\{a, b\}$ and $\{v, w\}$ as proof that $\mathcal{B}$ is not outerplanar, and stop.
9: **end if**
10: **end if**
11: **end for**
12: Report the embedding $\hat{\mathcal{B}}$ of $\mathcal{B}$ represented by list $\mathcal{A}$ as proof for the outerplanarity of $\mathcal{B}$.

**Algorithm 7.5**
Test for intersecting diagonals.

intersect, vertices $a, b, c, d$ are pairwise distinct. Moreover, edges $\{a, b\}$ and $\{c, d\}$ cannot intersect if intervals $[\nu(a), \nu(b)]$ and $[\nu(c), \nu(d)]$ are disjoint or one is contained in the other because in both cases, vertices $a$ and $b$ partition cycle $C$ into two paths so that vertices $c$ and $d$ are contained in the same path. Hence, w.l.o.g., $\nu(a) < \nu(c) < \nu(b) < \nu(d)$. But then the algorithm would push edge $\{a, b\}$ before edge $\{c, d\}$ on stack $S$ and try to pop edge $\{a, b\}$ from stack $S$ before popping edge $\{c, d\}$. Thus, it reports that graph $\mathcal{B}$ is not outerplanar. Note, however, that the algorithm does not necessarily report edges $\{a, b\}$ and $\{c, d\}$ as proof for the non-outerplanarity of $\mathcal{B}$ because it may detect another conflict before finding the intersection between $\{a, b\}$ and $\{c, d\}$. Next we show that if the algorithm reports a pair of edges as proof for the non-outerplanarity of $G$, then the two reported edges are diagonals that intersect. This proves that no matter which pair of edges the algorithm reports, the

graph defined by cycle $C$ and the two reported edges is homeomorphic to $K_4$. It also shows that the algorithm does not report any conflicts if $\mathcal{B}$ is outerplanar because in this case there is no pair of intersecting diagonals.

So assume that the algorithm reports an intersection. That is, it tries to pop an edge $\{v, w\}$ from the top of the stack which is not on the top of the stack. Let $\{a, b\}$ be the edge on the top of the stack. Then $\nu(v) \leq \nu(a) \leq \nu(w) \leq \nu(b)$. Now observe that all these inequalities are strict. To see this, consider the three possibilities for equality:

If $\nu(v) = \nu(a)$, edges $\{a, b\}$ and $\{a, w\}$ appear in this order counterclockwise around $a$, so that edge $\{a, b\}$ is pushed on the stack before edge $\{a, w\}$ and hence cannot be above edge $\{v, w\}$ on $S$.

If $\nu(a) = \nu(w)$, edges $\{a, v\}$ and $\{a, b\}$ appear in this order counterclockwise around $a$, so that edge $\{a, v\}$ is popped from the stack before edge $\{a, b\}$ is pushed on the stack. Hence, edge $\{a, b\}$ cannot be on the top of the stack when the algorithm tries to remove edge $\{a, v\}$ from the top of the stack.

Finally, if $\nu(w) = \nu(b)$, edges $\{a, b\}$ and $\{v, b\}$ appear in this order counterclockwise around $b$, so that edge $\{v, b\}$ is popped from the stack after edge $\{a, b\}$. This contradicts the assumption that edge $\{a, b\}$ is on the top of the stack when the algorithm tries to remove edge $\{v, b\}$ from the top of the stack.

This shows that $\nu(v) < \nu(a) < \nu(w) < \nu(b)$, so that edges $\{a, b\}$ and $\{v, w\}$ together with cycle $C$ define a subgraph of $G$ which is homeomorphic to $K_4$. □

The $K_{2,3}$-test during the construction of the boundary cycle can be incorporated in the embedding algorithm without increasing the I/O-complexity of that phase. Given list $\mathcal{A}$, the test for intersecting diagonals takes $\mathcal{O}(\text{scan}(|\mathcal{A}|)) = \mathcal{O}(\text{scan}(N))$ I/Os. To see this, observe that scanning $\mathcal{A}$ takes $\mathcal{O}(\text{scan}(N))$ I/Os. Every edge $\{v, w\}$, $\nu(v) < \nu(w)$, in $G$ is pushed on the stack at most once, namely when the traversal visits vertex $v$, and removed at most once, namely when the traversal visits vertex $w$. Thus, Algorithm 7.5 performs $\mathcal{O}(N)$ stack operations, which takes $\mathcal{O}(\text{scan}(N))$ I/Os. We have shown the following theorem.

**Theorem 7.1** *Given a graph $G = (V, E)$, it takes $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os and linear space to test whether $G$ is outerplanar and to provide proof for the decision of the algorithm by constructing an outerplanar embedding of $G$ or extracting a subgraph of $G$ which is homeomorphic to $K_{2,3}$ or $K_4$.*

## 7.2 Triangulation

Algorithms on outerplanar graphs are simplified if the given graph is triangulated. For instance our algorithms for BFS and SSSP presented in Section 7.4 assume that the input graph is triangulated. Hence, computing a triangulation of an outerplanar graph is an important algorithmic problem. In this section, we show that this problem can be solved in a linear number of I/Os for an embedded connected outerplanar graph. Our triangulation algorithm can easily be extended to deal with disconnected graphs as follows: On encountering a vertex $v$ which is the vertex with the smallest index $\nu(v)$ in its connected component, we add an edge $\{u, v\}$, $\nu(u) = \nu(v) - 1$ to $G$. This can be done on the fly while triangulating $G$ and transforms $G$ into a connected supergraph $G'$, whose triangulation is also a triangulation of $G$.

Formally, a *triangulation* of an outerplanar graph $G$ is a biconnected outerplanar supergraph $\Delta$ of $G$ with the same vertex set as $G$ and whose interior faces are triangles.[2] We show how to compute a list $\mathcal{D}$ representing the embedding $\hat{\Delta}$ of $\Delta$ from the list $\mathcal{A}$ representing the embedding $\hat{G}$ of $G$. In the rest of this section, we do not distinguish between a graph and its embedding. All graphs are considered to be embedded.

We need a few definitions to present our algorithm. An *ordered triangulation* of $G$ is a list $\mathcal{T}$ representing the dual tree $T$ of a triangulation $\Delta$ of $G$ with the following properties: (1) The vertices $v_1, \ldots, v_N$, where $\nu(v_i) < \nu(v_{i+1})$, for $1 \leq i < N$, appear

[2]This definition refers to the faces of $\Delta$ without explicitly referring to an embedding $\hat{\Delta}$ of $\Delta$ that defines these faces. For a planar graph $G$, its faces are well-defined only if an embedding $\hat{G}$ of $G$ is given. It is easy to show, however, that the outerplanar embedding of a triangulation as defined above is unique, except for flipping the whole graph (see Section 7.5).

in this order in a clockwise traversal of the outer boundary of $\Delta$. (2) A clockwise traversal of the boundary from $v_1$ to $v_N$ defines an Euler tour of $T$, which in turn defines a postorder numbering of the vertices in $T$. List $\mathcal{T}$ stores the vertices of $T$ sorted according to this postorder numbering.

Let $r \in G$ be the vertex with $\nu(r) = 1$. An *ordered partial triangulation* of $G$ w.r.t. the shortest monotone path $P = (r = v_0, \ldots, v_p = v)$ from vertex $r$ to some vertex $v$ is a list $\mathcal{T} = \mathcal{T}_1 \circ \cdots \circ \mathcal{T}_p$, where list $\mathcal{T}_i$ is an ordered triangulation of the subgraph of $G$ defined by all edges $\{a, b\} \in G$, $\nu(v_{i-1}) \leq \nu(a), \nu(b) \leq \nu(v_i)$. (Note that list $\mathcal{T}_i$ is empty if $\nu(v_{i-1}) + 1 = \nu(v_i)$.)

The *fringe* $\mathcal{F}(P)$ of a monotone path $P = (v_0, \ldots, v_p)$ in $G$ is a list of directed edges $\langle (v_0, w_{0,0}), \ldots, (v_0, w_{0,i_0}), (v_1, w_{1,0}), \ldots, (v_1, w_{1,i_1}), \ldots, (v_p, w_{p,0}), \ldots, (v_p, w_{p,i_p}) \rangle$, where for each $0 \leq j < p$, edges $\{v_j, w_{j,k}\}$, $0 \leq k \leq i_j$, are the edges in $G$ incident to $v_j$ with $\nu(v_{j+1}) \leq \nu(w_{j,k})$. For $v_p$, we require that $\nu(v_p) < \nu(w_{p,k})$. The edges incident to every vertex $v_j$ are sorted so that the path $P = (v_0, \ldots, v_j, w_k)$ is to the left of path $P = (v_0, \ldots, v_j, w_{k-1})$, for $0 < k \leq i_j$.

Our algorithm consists of two phases. The first phase (Algorithm 7.6) produces an ordered triangulation of $G$. A vertex $\alpha$ of $T$ is labelled with the vertices $u, v, w$ of $G$ in clockwise order along the boundary of the triangle represented by $\alpha$. Hence, we denote $\alpha$ as the vertex $(u, v, w)$. The second phase (Algorithm 7.7) uses list $\mathcal{T}$ to extract the list $\mathcal{D}$ representing the embedding $\hat{\Delta}$ of $\Delta$.

We say that Algorithm 7.6 *visits* vertex $v$ when the for-loop inspects the first edge $(v, w) \in \mathcal{A}$ (which is the first edge in $A(v)$).

**Lemma 7.10** *When Algorithm 7.6 visits vertex $v \in G$, the stack $S$ represents the fringe of the shortest monotone path $P$ in $G$ from $r$ to $u$, where $\nu(u) = \nu(v) - 1$. List $\mathcal{T}$ is an ordered partial triangulation of $G$ w.r.t. $P$.*

*Proof.* We prove this claim by induction on $\nu(v)$. If $\nu(v) = 1$, $v = r$ is the first vertex to be visited, so that $S$ is empty, and the claim of the lemma trivially holds. In this

**Procedure** OrderedTriangulation

**Input:** A list $\mathcal{A}$ representing the embedding of an outerplanar graph $G$.
**Output:** An ordered triangulation $\mathcal{T}$ of $G$.

```
 1: Initialize stack S to be empty.
 2: for each entry (v, w) ∈ A do
 3:   if the previous entry in A exists and is of the form (v', w'), v' ≠ v then
 4:     if ν(w) < ν(v) − 1 then
 5:                                                        {Bridge cutpoints.}
 6:       repeat
 7:         (a, b) ← Pop(S)
 8:         Append vertex (a, b, v) to T.
 9:       until a = w
10:     end if
11:   else
12:     if ν(v) < ν(w) then
13:       Push(S, (v, w))
14:     else
15:       Pop(S)                    {Triangulate the interior faces of G.}
16:       repeat
17:         (a, b) ← Pop(S)
18:         Append vertex (a, b, v) to T.
19:       until a = w
20:     end if
21:   end if
22: end for
23: Let v be the last vertex visited within the loop.   {Bridge remaining cutpoints.}
24: Pop(S)
25: while S is not empty do
26:   (a, b) ← Pop(S)
27:   Append vertex (a, b, v) to T.
28: end while
```

**Algorithm 7.6**
Computing the dual of the triangulation.

case, $\nu(w) > \nu(v)$, for all edges $(v, w) \in A(v)$. Thus, while inspecting $A(v)$, each iteration of the for loop executes Line 13, which pushes edge $(v, w)$ on the stack. By Observation 7.1, $\nu(w_1) > \nu(w_2) > \cdots > \nu(w_k)$, where $A(v) = \langle (v, w_1), \ldots, (v, w_k) \rangle$. Thus, the claim also holds for vertex $v'$ with $\nu(v') = 2$.

So assume that $\nu(v) > 2$ and that the claim holds for $u$, $\nu(u) = \nu(v) - 1$. By Observation 7.1, there exists an index $j$ such that $\nu(w_{j+1}) > \cdots > \nu(w_k) > \nu(u) > \nu(w_1) > \cdots > \nu(w_j)$, where $A(u) = \langle (u, w_1), \ldots, (u, w_k) \rangle$. We split the iterations inspecting $A(u)$ into three phases. The first phase inspects edge $(u, w_1)$. The second phase inspects edges $(u, w_2), \ldots, (u, w_j)$. And the third phase inspects edges $(u, w_{j+1}), \ldots, (u, w_k)$.

The iteration of the first phase executes Lines 4–10 of Algorithm 7.6. The iterations of the second phase execute Lines 15–19. The iterations of the third phase execute Line 13.

For the iteration of the first phase, vertex $w_1$ is a vertex on the path $P$ whose fringe is stored on $S$. To see this, observe that $P$ is a monotone path from $r$ to vertex $y$ with $\nu(y) = \nu(u) - 1$. If $w_1 = y$, we are done. So assume that $w_1 \neq y$. In this case, $\nu(w_1) < \nu(y)$ because $\nu(w_1) < \nu(u)$. Now observe that $G$ contains a monotone path $P'$ from $r$ to $w_1$, which can be extended to a monotone path $P''$ from $r$ to $u$ by appending edge $\{w_1, u\}$. Let $x$ be the last vertex on $P$ which is in $P \cap P''$. If $x = w_1$, then $w_1 \in P$. Otherwise, let $P''' = P(r, x) \circ P''(x, u)$. Now either $P$ is to the left of $P'''(r, w_1)$, or $P'''$ is to the left of $P$. In both cases, we obtain a contradiction to Lemma 7.8(i) because paths $P$, $P'''$, and $P'''(r, w_1)$ are monotone, and $\nu(w_1) < \nu(y) < \nu(u)$.

If $\nu(w_1) = \nu(u) - 1$, edge $(w_1, u)$ is the last edge in the fringe $\mathcal{F}(P)$ represented by the current stack. Assume that this is not the case. Then let $(w_1, z)$ be the edge succeeding $(w_1, u)$ in $\mathcal{F}(P)$, and let $P_1$ and $P_2$ be the paths obtained by appending edges $(w_1, z)$ and $(w_1, u)$ to $P$. Path $P_1$ is to the left of $P_2$; but $\nu(z) > \nu(u)$, which contradicts Lemma 7.8(ii). Thus, $S$ represents the fringe of a monotone path from $r$

to $u$ whose edges $\{u, w\}$, $\nu(w) > \nu(u)$ have been removed. $\mathcal{T}$ is an ordered partial triangulation of $G$ w.r.t. this path.

If $\nu(w_1) < \nu(u) - 1$, there is no edge in $\mathcal{F}(P)$ which is incident to a vertex succeeding $w_1$ along $P$ and is not part of $P$ itself. Assume the contrary. That is, there is an edge $(w_2, z) \in \mathcal{F}(P)$ such that $w_2$ succeeds $w_1$ along $P$. If $w_2 = y$, then $\nu(z) > \nu(y)$. But this implies that $\nu(z) > \nu(u)$ because $\nu(u) = \nu(y) + 1$. Thus, we obtain a contradiction to Lemma 7.8(i) because either $P \circ \{y, z\}$ is to the left of $P(r, w_1) \circ \{w_1, u\}$, or $P(r, w_1) \circ \{w_1, u\}$ is to the left of $P$. If $w_2 \neq y$, there are two cases. If $\nu(z) > \nu(y)$, we obtain a contradiction as in the case $w_2 = y$. Otherwise, let $w_3$ be the successor of $w_2$ along $P$. Then $P(r, w_3)$ is to the left of $P(r, w_2) \circ \{w_2, z\}$ because $\nu(z) > \nu(w_3)$. But this implies that $P$ is to the left of $P(r, w_2) \circ \{w_2, z\}$, thereby leading to a contradiction to Lemma 7.8(i) because $\nu(y) > \nu(z)$.

Thus, by adding an edge from $u$ to the vertex $y$ with $\nu(y) = \nu(u) - 1$ and triangulating the resulting face bounded by $P(w_1, y)$ and edges $\{y, u\}$ and $\{w_1, u\}$, as done in Lines 6–9 of Algorithm 7.6, a partial triangulation of $G$ w.r.t. the path $P'$ defined as the concatenation of $P(r, w_1)$ and edge $\{w_1, u\}$ is obtained. The triangulation is ordered, as the triangles are added from $y$ toward $w_1$ along $P$. Stack $S$ now represents the fringe $\mathcal{F}(P')$ of $P'$ after removing edges $\{u, w\}$, $\nu(w) > \nu(u)$.

For every edge $(u, w)$ inspected in the second phase, edge $(w, u)$ must be part of the shortened fringe of $P'$ represented by $S$. This can be shown using the same argument as the one showing that vertex $w_1$ in the iteration of the first phase is part of $P$. By Observation 7.1, the edges inspected during the second phase are inspected according to the order of their endpoints from $w_1$ toward $r$ along $P$. Using the same arguments as the ones showing that $(w_1, u)$ is the last edge in the fringe $\mathcal{F}(P)$ if $\nu(w_1) = \nu(u) - 1$, it can be shown that there cannot be any edges $\{a, b\}$, $\nu(w_j) < \nu(a)$ and $\nu(b) \neq \nu(u)$, in the fringe of $P'$, so that the top of stack $S$ represents the subpath $P'(w_j, u)$ with dangling edges $(w_j, u), \ldots, (w_2, u)$ attached. The iterations of the second phase now triangulate the faces defined by $P'(w_j, u)$ and edges $\{w_j, u\}, \ldots, \{w_2, u\}$, so that at the end of this phase, stack $S$ represents a monotone path $P''$ from $r$ to $u$, and $\mathcal{T}$

represents an ordered partial triangulation of $G$ w.r.t. path $P''$. We argue as follows that path $P''$ is the *shortest* monotone path from $r$ to $u$ in $G$:

If there were a shorter monotone path $Q$ from $r$ to $u$, this path would have to pass through one of the bicomps of the partial triangulation represented by $\mathcal{T}$ or through one of the vertices not inspected yet. The latter would result in a non-monotone path, as for each such vertex $x$, $\nu(x) > \nu(u)$. In the former case, if $Q$ passes through the bicomp defined by vertices $z \in G$, where $\nu(x) \leq \nu(z) \leq \nu(y)$ and $\{x, y\}$ is an edge in $P''$, replacing the subpath $Q(x, y)$ by edge $\{x, y\}$ in $Q$ would result in a shorter path $Q'$. Thus, $P''$ is indeed the shortest monotone path from $r$ to $u$.

The iterations of the third phase finally add edges $\{u, w_{j+1}\}, \ldots, \{u, w_k\}$ to the stack $S$, so that the representation of the fringe $\mathcal{F}(P'')$ of $P''$ is complete, and the claim holds for $v$ as well. □

After the for-loop is finished inspecting all edges in $\mathcal{A}$, the stack $S$ represents the fringe $\mathcal{F}(P)$ of the shortest monotone path $P$ in $G$ from $r$ to the last vertex $v$ with $\nu(v) = N$, and $\mathcal{T}$ represents an ordered partial triangulation of $G$ w.r.t. $P$. In this case, $\mathcal{F}(P) = P$, as the adjacency lists of all vertices of $G$ have been inspected. Vertices $v$ and $r$ are not necessarily adjacent, so that the interior vertices of $P$ are cutpoints of the partial triangulation constructed so far. To complete the triangulation, the while-loop in Lines 24–28 makes vertex $v$ adjacent to vertex $r$ and triangulates the resulting face. Thus, Algorithm 7.6 does indeed produce an ordered triangulation of $G$.

The following lemma shows that Algorithm 7.7 correctly constructs an embedding of the triangulation $\Delta$ which is represented by the list $\mathcal{T}$ computed by Algorithm 7.6. For a triangle $(a, b, c) \in \mathcal{T}$, let $\tau((a, b, c))$ be its position in $\mathcal{T}$. It follows from the fact that $\mathcal{T}$ represents a postorder traversal of tree $T$ that triangles $(a, b, c)$ with $\tau((a, b, c)) \geq j$, for some integer $1 \leq j \leq |\mathcal{T}|$, represent a subgraph $\Delta_j$ of $\Delta$ which is a triangulation. Let $\bar{\Delta}_j$ be the subgraph of $\Delta$ induced by all triangles $(a, b, c)$, $\tau((a, b, c)) < j$. Denote the set of edges shared by $\Delta_j$ and $\bar{\Delta}_j$ by $\partial\Delta_j$.

**Procedure** ExtractTriangulation

**Input:** An ordered triangulation $\mathcal{T}$ of $G$.
**Output:** A list $\mathcal{D}$ representing the embedding $\hat{\Delta}$ of the triangulation $\Delta$ represented by $\mathcal{T}$.

```
1: Initialize stack S to be empty.
2: Initialize list D to be empty.
3: for each vertex (a, b, c) ∈ T in reverse order do
4:   if (a, b, c) is the first visited vertex then
5:     {Assume that ν(a) < ν(b) < ν(c).}
6:     Prepend entry (c, a) to D.
7:     Prepend entry (c, b) to D.
8:     Push(S, (a, c))
9:     Push(S, (a, b))
10:    Push(S, (b, a))
11:    Push(S, (b, c))
12:  else
13:    while the top of the stack S does not equal (b, a), (c, b) or (a, c) do
14:      (d, e) ← Pop(S)
15:      Prepend entry (d, e) to D.
16:    end while
17:    {Assume w.l.o.g. that the top of the stack is (b, a).}
18:    Prepend entry (a, c) to D.
19:    Push(S, (b, c))
20:    Push(S, (c, b))
21:    Push(S, (c, a))
22:  end if
23: end for
24: while S is not empty do
25:   (a, b) ← Pop(S)
26:   Prepend entry (a, b) to D.
27: end while
```

**Algorithm 7.7**
Extracting the embedding $\hat{\Delta}$ of $\Delta$ from the tree $T$.

**Lemma 7.11** *After processing triangle $(a,b,c) \in \mathcal{T}$ with $\tau((a,b,c)) = j$, the concatenation of $S$ and $\mathcal{D}$ represents an embedding of $\Delta_j$. For every edge $\{x,y\} \in \partial\Delta_j$, stack $S$ stores an entry $(x,y)$, where $x$ and $y$ appear clockwise around the only triangle in $\Delta_j$ containing both $x$ and $y$.*

*Proof.* We prove the claim by induction on $j$. If $j = |\mathcal{T}|$, triangle $(a,b,c)$ is the first visited triangle. Thus, Lines 5–11 of Algorithm 7.7 are executed. The concatenation of $S$ and $\mathcal{D}$ is of the form $\langle (a,c),(a,b),(b,a),(b,c),(c,b),(c,a)\rangle$, where $\nu(a) = 1$ and $\nu(c) = |G|$. This represents an embedding of triangle $(a,b,c)$. Moreover, stack $S$ stores edges $(a,b)$ and $(b,c)$, and edge $\{a,c\}$ cannot be shared by $\Delta_j$ and $\bar{\Delta}_j$ because edge $\{a,c\}$ is a boundary edge of $\Delta$.

So assume that the claim holds for $j > k$. We prove the claim for $j = k$. In this case, Lines 13–21 are executed. By the induction hypothesis, the concatenation of $S$ and $\mathcal{D}$ represents an embedding of $\Delta_{k+1}$, and $S$ stores edge $(b,a)$, where edge $\{a,b\}$ is shared by triangle $(a,b,c)$ and triangulation $\Delta_{k+1}$. The while-loop in Lines 13–16 transfers edges from $S$ to $\mathcal{D}$ until edge $(b,a)$ is on the top of the stack. Lines 18–21 "insert" vertex $c$ into the boundary cycle of $\Delta_{k+1}$, by inserting entries $(b,c)$, $(c,b)$, $(c,a)$, and $(a,c)$ between entries $(b,a)$ and $(a,b)$ in the sequence represented by $S$ and $\mathcal{D}$. The result is an embedding of $\Delta_k$.

The edges removed from the stack during the while-loop in Lines 13–16 cannot be in $\partial\Delta_k$, as for every triangle $(a',b',c')$ sharing one of those edges with $\Delta_{k+1}$, $\tau((a',b',c')) > \tau((a,b,c))$. This follows from the fact that $\mathcal{T}$ is an ordered triangulation. Edge $\{a,b\}$ cannot be in $\partial\Delta_k$, as it is already shared by $\Delta_{k+1}$ and triangle $(a,b,c)$. Thus, every edge in $\partial\Delta_k$ is either shared by $\Delta_{k+1}$ and $\bar{\Delta}_k$ or by triangle $(a,b,c)$ and $\bar{\Delta}_k$. We have just argued that the former edges are not removed from $S$, and the latter edges can only be edges $\{a,c\}$ or $\{b,c\}$, whose representations have been put on the stack. Thus, the claim holds for $j = k$ as well. □

Lemma 7.11 implies that after the for-loop has inspected all triangles in $\mathcal{T}$, the concatenation of $S$ and $\mathcal{D}$ represents an outerplanar embedding of $\Delta$. Thus, after prepending the entries in $S$ to $\mathcal{D}$ as done in Lines 24–27 of Algorithm 7.7, list $\mathcal{D}$

represents an embedding of $\Delta$. This shows that Algorithms 7.6 and 7.7 correctly compute an outerplanar embedding $\hat{\Delta}$ of a triangulation $\Delta$ of $G$.

**Theorem 7.2** *Given a list $\mathcal{A}$ representing an embedding $\hat{G}$ of a connected outerplanar graph $G$ with $N$ vertices, it takes $\mathcal{O}(\text{scan}(N))$ I/Os and linear space to compute a list $\mathcal{D}$ representing an outerplanar embedding $\hat{\Delta}$ of a triangulation $\Delta$ of $G$.*

*Proof.* List $\mathcal{D}$ can be computed using Algorithms 7.6 and 7.7. The correctness of this procedure follows from Lemmas 7.10 and 7.11.

To prove the I/O-bound of our algorithm, observe that Algorithm 7.6 scans list $\mathcal{A}$ and writes list $\mathcal{T}$. In addition it performs some stack operations. Scanning list $\mathcal{A}$ takes $\mathcal{O}(\text{scan}(|\mathcal{A}|)) = \mathcal{O}(\text{scan}(N))$ I/Os. Writing list $\mathcal{T}$ takes $\mathcal{O}(\text{scan}(|\mathcal{T}|)) = \mathcal{O}(\text{scan}(N))$ I/Os. The number of stack operations performed by Algorithm 7.6 is twice the number of PUSH operations it performs. However, every entry of list $\mathcal{A}$ causes at most one PUSH operation to be performed, so that Algorithm 7.6 performs $\mathcal{O}(|\mathcal{A}|) = \mathcal{O}(N)$ stack operations, which takes $\mathcal{O}(\text{scan}(N))$ I/Os.

Algorithm 7.7 scans list $\mathcal{T}$ and writes list $\mathcal{D}$. As both lists have size $\mathcal{O}(N)$, this takes $\mathcal{O}(\text{scan}(N))$ I/Os. The number of stack operations performed by Algorithm 7.7 is $\mathcal{O}(|\mathcal{T}|) = \mathcal{O}(N)$, as each entry in $\mathcal{T}$ causes at most four PUSH operations to be performed. Thus, Algorithm 7.7 takes $\mathcal{O}(\text{scan}(N))$ I/Os as well. □

## 7.3 Computing Separators of Outerplanar Graphs

In this section, we discuss the problem of finding a small $\varepsilon$-separator of an outerplanar graph. We show the following result.

**Theorem 7.3** *Given an embedded outerplanar graph $G = (V, E)$ with $N$ vertices, represented as a list $\mathcal{A}$, a weight function $\omega : V \to \mathbb{R}_0^+$, and a constant $0 < \varepsilon < 1$, it takes $\mathcal{O}(\text{scan}(N))$ I/Os and linear space to compute an $\varepsilon$-separator $S$ of size $\mathcal{O}(1/\varepsilon)$ for $G$.*

We assume that graph $G$ is triangulated and that the vertices of its dual tree $T$ are given, sorted according to some preorder numbering of $T$. This information can be computed in $\mathcal{O}(\text{scan}(N))$ I/Os using the triangulation algorithm of Section 7.2. Every separator of a triangulation of $G$ is also a separator of $G$. Given an edge $e \in T$ whose removal partitions $T$ into two trees $T_1$ and $T_2$, trees $T_1$ and $T_2$ represent two subgraphs $G_1$ and $G_2$ of $G$ such that $G_1$ and $G_2$ share a pair of vertices, $v$ and $w$. Let $e^* = \{v, w\}$ be the edge dual to edge $e \in T$. The connected components of $G - \{v, w\}$ are graphs $G_1 - \{v, w\}$ and $G_2 - \{v, w\}$.

We choose a degree-1 vertex $r$ of $T$ as the root of $T$. For a vertex $v \in T$, let $\Delta(v)$ be the triangle of $G$ represented by $v$. Let $V_v$ be the vertex set of $\Delta(v)$. Let $T(v)$ be the subtree of $T$ rooted at $v$, and let $G(v)$ be the subgraph of $G$ defined as the union of all triangles $\Delta(w)$, $w \in T(v)$. That is, $T(v)$ is the dual tree of $G(v)$. If $v \neq r$, let $p(v)$ be the parent of $v$ in $T$ and $e_v = \{v, p(v)\}$ be the edge connecting $v$ to its parent $p(v)$. Then $e_v^* = \{x_v, y_v\}$ is the edge of $G$ dual to edge $e_v$.

Our algorithms proceeds in three phases. The first phase of the algorithm computes weights $\omega(v)$ for the vertices of $T$ so that $\omega(T) = \omega(G)$, and for $v \neq r$, $\omega(T(v)) = \omega(G(v) - \{x_v, y_v\})$. The second phase of the algorithm computes a small edge-separator of $T$ w.r.t. these vertex weights. The third phase of the algorithm computes the corresponding vertex separator of $G$. Next we discuss these three phases in detail.

**Phase 1: Computing the weights of the dual vertices.** We define weights $\omega(v)$ as follows. For the root $r$ of $T$, let $\omega(r) = \omega(\Delta(r))$. For every other vertex $v \in T$, let $\omega(v) = \omega(z_v)$, for $z_v \in V_v \setminus \{x_v, y_v\}$. Note that $z_v$ is unique. Given that the vertices of $T$ are stored in postorder w.r.t. root $r$, these vertex weights can be computed in $\mathcal{O}(\text{scan}(N))$ I/Os by processing $T$ top-down using time-forward processing, since $V_v$ is stored with $v$ in $T$, and $\{x_v, y_v\} = V_v \cap V_{p(v)}$. To arrange the vertices in this order, we use procedure RootTree (Algorithms 7.8 and 7.9 on pages 106 and 107). The following lemma is easy to show by induction on the size of tree $T(v)$.

**Lemma 7.12** *The weight of tree $T$ is $\omega(T) = \omega(G)$. For every vertex $v \neq r$, $\omega(T(v)) = \omega(G(v) - \{x_v, y_v\})$.*

**Phase 2: Computing a small edge-separator of $T$.** The next step of the algorithm is to compute a set $C$ of edges of $T$ so that none of the connected components $T_0, T_1, \ldots, T_k$ of $T - C$ has weight exceeding $\varepsilon$, except possibly $T_0$. Component $T_0$ is the one containing the root $r$ of $T$. If $\omega(T_0) > \varepsilon$, then $T_0 = (\{r\}, \emptyset)$. At this point we have to make the assumption that no vertex in $T$, except $r$, has weight exceeding $\varepsilon/2$. We show how to ensure this condition when discussing Phase 3 of our algorithm.

In order to compute $C$, we process $T$ bottom-up and apply the following rules: When visiting a leaf $v$ of $T$, we define $\omega'(v) = \omega(v)$. At an internal node $v \neq r$ of $T$ with children $w_1$ and $w_2$, we proceed as follows: If one of the children, say $w_2$, does not exist, we define $\omega'(w_2) = 0$. If $\omega(v) + \omega'(w_1) + \omega'(w_2) \leq \varepsilon/2$, let $\omega'(v) = \omega(v) + \omega'(w_1) + \omega'(w_2)$. If $\varepsilon/2 < \omega(v) + \omega'(w_1) + \omega'(w_2) \leq \varepsilon$, we define $\omega'(v) = 0$ and add edge $\{v, p(v)\}$ to $C$. If $\varepsilon < \omega(v) + \omega'(w_1) + \omega'(w_2)$, we define $\omega'(v) = 0$ and add edges $\{v, p(v)\}$ and $\{v, w_1\}$ to $C$. If $v = r$, $v$ has a single child $w$. If $\omega(v) + \omega'(w) > \varepsilon$, we add edge $\{v, w\}$ to $C$.

This procedure takes $\mathcal{O}(\text{scan}(N))$ I/Os using time-forward processing. Instead of producing list $C$ explicitly, we label every edge in $T$ as either being contained in $C$ or not. This representation simplifies Phase 3. The following two lemmas show that $C$ is almost an $\varepsilon$-edge-separator of size $\lfloor 2/\varepsilon \rfloor$ for $T$.

**Lemma 7.13** *Let $T_0, \ldots, T_k$ be the connected components of $T - C$, and let $r \in T_0$. Then $\omega(T_i) \leq \varepsilon$, for $1 \leq i \leq k$. If $\omega(T_0) > \varepsilon$, then $T_0 = (\{r\}, \emptyset)$.*

*Proof.* For every vertex $v \in T$, let $T_v$ be the component $T_i$ such that $v \in T_i$. We show that $\omega(T(v) \cap T_v) \leq \varepsilon$, for all vertices $v \neq r$ in $T$. This implies that $\omega(T_i) \leq \varepsilon$, for $1 \leq i \leq k$ because for the root $r_i$ of $T_i$, $T(r_i) \cap T_i = T_i$.

In order to prove this claim, we show a somewhat stronger result. In particular, we show that $\omega(T(v) \cap T_v) \leq \varepsilon$ if $v$ is the root of $T_v$, and $\omega'(v) = \omega(T(v) \cap T_v) \leq \varepsilon/2$ if $v$ is not the root of $T_v$. We prove this claim by induction on the size of tree $T(v)$.

If $|T(v)| = 1$, $v$ is a leaf, and $T(v) \cap T_v = (\{v\}, \emptyset)$, so that $\omega(T(v) \cap T_v) = \omega(v) \leq \varepsilon/2$ and $\omega'(v) = \omega(v)$.

If $|T(v)| > 1$, $v$ is an internal vertex with children $w_1$ and $w_2$. If $T_v = T_{w_1} = T_{w_2}$, then neither of $w_1$ and $w_2$ is the root of $T_v$. By the induction hypothesis, this implies that $\omega'(w_1) = \omega(T(w_1) \cap T_v) \leq \varepsilon/2$ and $\omega'(w_2) = \omega(T(w_2) \cap T_v) \leq \varepsilon/2$. This implies that $\omega(T(v) \cap T_v) = \omega(v) + \omega'(w_1) + \omega'(w_2)$. If $\omega(T(v) \cap T_v) > \varepsilon$, edge $\{v, w_1\}$ would have been added to $C$, contradicting the assumption that $T_v = T_{w_1}$. Thus, $\omega(T(v) \cap T_v) \leq \varepsilon$. If $\omega(T(v) \cap T_v) > \varepsilon/2$, our algorithm adds edge $\{v, p(v)\}$ to $C$, so that $v$ is the root of $T_v$. Otherwise, $\omega'(v) = \omega(v) + \omega'(w_1) + \omega'(w_2) = \omega(T(v) \cap T_v)$.

If $T_v = T_{w_2} \neq T_{w_1}$, $w_2$ is not the root of $T_v$. Thus, $\omega'(w_2) = \omega(T(w_2) \cap T_v) \leq \varepsilon/2$. This immediately implies that $\omega(T(v) \cap T_v) = \omega(v) + \omega(T(w_2) \cap T_v) \leq \varepsilon$. If $\omega(T(v) \cap T_v) > \varepsilon/2$, edge $\{v, p(v)\}$ is added to $C$, so that $v$ is the root of $T_v$. Otherwise, $\omega'(v) = \omega(v) + \omega'(w_1) + \omega'(w_2) = \omega(v) + \omega'(w_2) = \omega(T(v) \cap T_v)$.

Finally, if $T_v$, $T_{w_1}$, and $T_{w_2}$ are all distinct, vertices $w_1$ and $w_2$ are the roots of trees $T_{w_1}$ and $T_{w_2}$, so that our algorithm sets $\omega'(w_1) = \omega'(w_2) = 0$. This implies that $\omega'(v) = \omega(v) = \omega(T(v) \cap T_v) \leq \varepsilon/2$.

In order to show that $T_0 = (\{r\}, \emptyset)$ if $\omega(T_0) > \varepsilon$, we argue as follows: Let $w$ be the child of $r$ in $T$. If $T_r = T_w$ then vertex $w$ is not the root of $T_w$, so that $\omega'(w) = \omega(T(w) \cap T_r)$. If $\omega(T_r) = \omega(r) + \omega'(w) > \varepsilon$, our algorithm would have added edge $\{r, w\}$ to $C$. Thus, $\omega(T_r) \leq \varepsilon$ if $T_r = T_w$. If $T_r \neq T_w$, $T_r = (\{r\}, \emptyset)$, as $w$ is the only child of $r$. □

**Lemma 7.14** $|C| \leq \lfloor 2\omega(T)/\varepsilon \rfloor$.

*Proof.* In order to prove the lemma, we charge the edges in $C$ to individual subtrees $T_i$ of $T - C$ or to pairs of subtrees $\{T_i, T_j\}$ of $T - C$. Every subtree $T_i$ is charged at most once, either individually or as part of a pair of subtrees. Every individual tree that is being charged has weight at least $\varepsilon/2$ and is being charged for one edge in $C$. Every pair of trees that is being charged has weight at least $\varepsilon$ and is being charged for two edges in $C$. Thus, on average, we charge at most one edge per $\varepsilon/2$ units of weight. Hence, $|C| \leq \lfloor 2\omega(T)/\varepsilon \rfloor$. It remains to show how to distribute the charges.

Consider the way edges in $C$ are added to $C$. An edge $\{v, p(v)\}$ that is being added to $C$ while processing $v$ is added either alone or along with an edge $\{v, w_1\}$, where $w_1$ is a child of $v$. In the former case, $\varepsilon/2 < \omega(T_v) \leq \varepsilon$, and we charge edge $\{v, p(v)\}$ to graph $T_v$. If edge $\{v, p(v)\}$ is added to $C$ along with edge $\{v, w_1\}$, $\varepsilon < \omega(T_v) + \omega(T_{w_1})$. Then we charge these two edges to the pair $\{T_v, T_{w_1}\}$. Every subtree $T_i$ with root $r_i$ is charged only for edge $\{r_i, p(r_i)\}$. Thus, every subtree $T_i$ is charged at most once. If edge $\{r, w\}$ is being added to $C$ while processing the child $w$ of the root $r$, then this edge is already covered by this charging scheme. Otherwise, edge $\{r, w\}$ is being added because $\omega(r) + \omega'(w) > \varepsilon$. In this case, we charge the edge to $T_0$. Component $T_0$ is never charged for any other edges because its root does not have a parent. □

**Phase 3: Computing the vertex-separator of $G$.** In order to compute an $\varepsilon$-separator of $G$, we first have to ensure that no vertex, except possibly the root, in the dual tree $T$ has weight more than $\varepsilon/2$. To ensure this, it is sufficient to guarantee that no vertex in $G$ has weight exceeding $\varepsilon/2$ because every vertex in $T$, except the root obtains its weight from a single vertex in $G$. Thus, the vertex separator is computed as the union of two sets $S_1$ and $S_2$. Set $S_1$ contains all vertices of weight more than $\varepsilon/2$ in $G$. Then we set $\omega(v) = 0$, for each vertex $v \in S_1$ and compute the edge separator $C$ of $T$ w.r.t. these modified vertex weights. Every edge $e \in C$ corresponds to an edge $e^* = \{x, y\}$ in $G$. Vertices $x$ and $y$ are added to $S_2$. By Lemma 7.14,

$$\begin{aligned} |C| &\leq \left\lfloor \frac{2\omega(T)}{\varepsilon} \right\rfloor \\ &= \left\lfloor \frac{2(\omega(G) - \omega(S_1))}{\varepsilon} \right\rfloor \\ &\leq \left\lfloor \frac{2(\omega(G) - \frac{\varepsilon}{2}|S_1|)}{\varepsilon} \right\rfloor \\ &= \left\lfloor \frac{2\omega(G)}{\varepsilon} \right\rfloor - |S_1|. \end{aligned}$$

Thus, $|S_2| \leq 4\omega(G)/\varepsilon - 2|S_1|$, so that $|S| \leq |S_1| + |S_2| \leq 4\omega(G)/\varepsilon$. The computation of $S_1$ takes $\mathcal{O}(\text{scan}(N))$ I/Os. Given $C$, set $S_2$ is easily computed in $\mathcal{O}(\text{scan}(N))$ I/Os using a preorder traversal of $T$. We have to show that $S$ is an $\varepsilon$-separator of $G$.

**Lemma 7.15** *Vertex set $S$ is an $\varepsilon$-separator of $G$.*

*Proof.* Let $T_0, \ldots, T_k$ be the connected components of $T - C$. Let $G_0, \ldots, G_k$ be the subgraphs of $G$ such that $G_i$ is the union of all triangles $\Delta(v)$, $v \in T_i$. We show that every connected component of $G - S$ is completely contained in a subgraph $G_i$ and that $\omega(G_i - S) \leq \varepsilon$.

The first claim is easy to see. Indeed, all edges $e = \{v, w\} \in T$, $v \in T_i$, $w \notin T_i$, are in $C$, so that the endpoints of their dual edges are in $S$, and there is no path from a vertex not in $G_i$ to a vertex in $G_i$ that does not contain a vertex in $S$.

In order to prove the second claim, observe that for $i = 0$, $\omega(T_0) = \omega(G_0)$, by the definition of weights $\omega(v)$, $v \in T$. If $\omega(T_0) \leq \varepsilon$, then $\omega(G_0 - S) \leq \varepsilon$. Otherwise, $T_0 = (\{r\}, \emptyset)$, and $G_0 - S$ contains at most one vertex, whose weight is no more than $\varepsilon/2$.

For $i > 0$, let $r_i$ be the root of tree $T_i$. Then $\omega(T_i) = \omega(G_i - \{x_{r_i}, y_{r_i}\}) \geq \omega(G_i - S)$, as $x_{r_i}, y_{r_i} \in S$. But, $\omega(T_i) \leq \varepsilon$, by Lemma 7.13. □

## 7.4 DFS, BFS, and Single Source Shortest Paths

The problem of computing a DFS-tree of an embedded outerplanar graph $G$ can easily be solved in $\mathcal{O}(\text{scan}(N))$ I/Os, provided that the choice of the root of the tree is left to the algorithm: We choose the vertex $r$ with $\nu(r) = 1$ as the root of the tree. A DFS-tree with this root is already encoded in the list $\mathcal{A}$ representing the embedding of $G$ and can easily be extracted in $\mathcal{O}(\text{scan}(N))$ I/Os. If the DFS-tree has to have a particular root $r$, a simple stack algorithm can be used to extract the desired DFS-tree from the embedding of $G$.

**Theorem 7.4** *Given list $\mathcal{A}$ representing an outerplanar embedding of a connected outerplanar graph $G$ with $N$ vertices, a DFS-tree $T$ for $G$ can be constructed in $\mathcal{O}(\text{scan}(N))$ I/Os using linear space.*

In the rest of this section, we present an algorithm to solve the single source shortest path problem for an embedded connected outerplanar graph $G$ in a linear number of I/Os. Since breadth-first search is the same as the single source shortest path problem after assigning unit weights to the edges of $G$, this algorithm can also be used to compute a BFS-tree for $G$. We describe our algorithm assuming that graph $G$ is undirected. However, it is easy to verify that it generalizes in a straightforward manner to the directed case.

The first step of the algorithm is to triangulate the given graph $G$. Let $\Delta$ be the resulting triangulation. The additional edges in $\Delta$ are given infinite weight, so that the shortest path between two vertices in $\Delta$ is the same as the shortest path between these two vertices in $G$. The triangulation algorithm of Section 7.2 can easily be augmented to maintain edge weights. Recall that the triangulation algorithm first computes a list $\mathcal{T}$ of the triangles in $\Delta$ sorted according to a postorder traversal of the dual tree $T$ of $\Delta$. This representation of $\Delta$ is more useful for our SSSP-algorithm than the list $\mathcal{D}$ representing the embedding of $\Delta$.

Let the weight of an edge $e$ be denoted by $\omega(e)$. Given a source vertex $s$, we compute a shortest path tree of $G$ rooted at $s$ as follows: We choose a root vertex $s'$ of $T$ so that the triangle $\Delta(s')$ has vertex $s$ as one of its vertices. For every edge $e$ of $T$, let $T_e$ be the subtree of $T$ induced by all vertices $v$ so that the path from $s'$ to $v$ in $T$ contains edge $e$. That is, intuitively, tree $T_e$ is connected to the rest of $T$ through edge $e$. Let $\bar{T}_e$ be the subtree $T - T_e$ of $T$. Let $G_e$ be the union of triangles $\Delta(v)$, for all vertices $v \in T_e$. Graph $\bar{G}_e$ is defined analogously for $\bar{T}_e$. Then $G = G_e \cup \bar{G}_e$ and $G_e \cap \bar{G}_e = (\{x_e, y_e\}, \{e^*\})$. That is, $G_e$ and $\bar{G}_e$ share only the dual edge $e^*$ of $e$. The endpoints $x_e$ and $y_e$ of $e^*$ form a separator of $G$. Any simple path $P$ from $s$ to a vertex $v \in \bar{G}_e$ either does not contain a vertex of $G_e - \{x_e, y_e\}$, or it contains both $x_e$ and $y_e$. These observations suggest the following strategy:

First we compute weights $\omega'(e)$ for the edges of $G$. If there exists an edge $e^* \in T$ such that $e$ is dual to $e^*$, $\omega'(e)$ is defined as the length of the shortest path in $G_{e^*}$ between the endpoints $x_{e^*}$ and $y_{e^*}$ of $e$. Otherwise, $\omega'(e) = \omega(e)$. In the second step, let $x_v$ be the vertex of $T$ closest to $s$ so that $v \in \Delta(x_v)$, for all $v \in G$. Given that $x_v$ has children $w_1$ and $w_2$, we compute the distance $d'(s, v)$ in the graph $\bar{G}_{\{x_v,w_1\}} \cap \bar{G}_{\{x_v,w_2\}}$ w.r.t. the weights $\omega'(e)$ computed in the first step. As we show below, $d'(s, v) = d_G(s, v)$. At the end of this section, we show how to augment the algorithm to compute a shortest path tree.

**Rooting $T$.** In order to be able to perform the remaining steps of the algorithm in a linear number of I/Os using time-forward processing, we have to ensure that tree $T$ is rooted at a vertex $s'$ such that the source $s$ of the shortest path computation is a vertex of $\Delta(s')$, and that the vertices of $T$ are sorted according to a preorder (or postorder) traversal of $T$. After applying the triangulation algorithm of Section 7.2, the vertices of $T$ are stored in postorder, but the current root of $T$ may be different from $s'$.

As the reversal of a postorder numbering of $T$ is a preorder numbering of $T$, we assume w.l.o.g. that the vertices of $T$ are stored in preorder. The first step of our procedure to root $T$ at vertex $s'$ extracts an Euler-tour of $T$. Let $e_1, \ldots, e_k$ be the list of edges in the Euler tour. Then the tour is represented by a list $\mathcal{E}$ containing the source vertices of edges $e_1, \ldots, e_k$ in order. List $\mathcal{E} = \langle x_1, \ldots, x_t \rangle$ is transformed into a list $\mathcal{E}' = \langle x_k \ldots, x_t, x_1, \ldots, x_{k-1} \rangle$, where $x_k = s'$. List $\mathcal{E}'$ represents an Euler tour of $T$ starting at vertex $s'$. The final step of the algorithm extracts the vertices of $T$ in preorder. Algorithms 7.8 and 7.9 provide the details of this step.

In order to show the correctness of procedure RootTree, we show that the list $\mathcal{E}$ produced by Algorithm 7.8 describes an Euler tour of $T$ starting at the current root $r$ of $T$. This implies that list $\mathcal{E}'$ represents an Euler tour of $T$ starting at $s'$. Using this fact, we show that Algorithm 7.9 produces a list $\mathcal{T}'$ storing the vertices of $T$, when rooted at $s'$, in preorder.

**Procedure** RootTree

**Input:** A list $\mathcal{T}$ storing the vertices of the dual tree $T$ of $\Delta$ rooted at vertex $r$ with $\nu(r) = 1$ in preorder, and a vertex $s' \in T$.

**Output:** A list $\mathcal{T}'$ storing the vertices of the dual tree $T$ of $\Delta$ rooted at vertex $s'$ in preorder.

```
 1: Initialize stack S to be empty.
 2: for each vertex v ∈ T do
 3:   if v is the first vertex (i.e., the current root of T) then
 4:     Push(S, v)
 5:     Append v to E.
 6:   else
 7:     while the top of stack S is not equal to p(v) do
 8:       Pop(S)
 9:       Append the top of stack S to E.
10:     end while
11:     Push(S, v)
12:     Append v to E.
13:   end if
14: end for
15: while S is not empty do
16:   Pop(S)
17:   if S is not empty then
18:     Append the top of stack S to E.
19:   end if
20: end while
21: Remove the last element from E.
```

**Algorithm 7.8**
Rooting tree $T$ at vertex $s'$. (Part I: Computing an Euler tour.)

**Lemma 7.16** *List $\mathcal{E}$, as computed by Lines 1–21 of procedure* RootTree *(Algorithm 7.8), describes an Euler tour of $T$, starting at vertex $r$.*

*Proof.* In this proof, we refer to each vertex $v \in T$ by its preorder number. In order to prove the lemma, we show that the for-loop of Lines 2–14 maintains the following invariant: Stack $S$ stores the vertices on the path from the root $r$ to the current vertex $v$. List $\mathcal{E}$ represents a traversal of $T$ from $s$ to $v$ so that for all vertices $u < v$, $u \notin S$, edge $\{u, p(u)\}$ is traversed twice, once in each direction, and

**Procedure** RootTree (Continued)

22: Scan $\mathcal{E}$ to find the first index $k$ in list $\mathcal{E} = \langle x_1, \ldots, x_t \rangle$ such that $x_k = s'$. While scanning append elements $x_1, \ldots, x_{k-1}$ to a queue $Q$ and delete them from $\mathcal{E}$. Once $x_k$ has been found, move elements $x_1, \ldots, x_{k-1}$ from $Q$ to the end of the resulting list $\langle x_k, \ldots, x_t \rangle$. Call the resulting list $\mathcal{E}'$.
23: **for** each element $x_i \in \mathcal{E}'$ **do**
24: **if** $x_i$ is the first element in $\mathcal{E}'$ **then**
25: Append the pair $(x_i, \textbf{null})$ to list $\mathcal{T}'$. $\{x_i$ is the root of $T.\}$
26: **else if** $x_i$ is not equal to the top of stack $S$ **then**
27: Append the pair $(x_i, x_{i-1})$ to list $\mathcal{T}'$. $\{x_{i-1}$ is the parent of $x_i$ in the tree $T$ rooted at $s'.\}$
28: **if** the next element $x_{i+1} \neq p(x_i)$ **then**
29: Push$(S, x_i)$
30: **end if**
31: **else if** the next element $x_{i+1} = p(x_i)$ **then**
32: Pop$(S)$
33: **end if**
34: **end for**

---

**Algorithm 7.9**
Algorithm 7.8 continued. (Part II: Computing the new preorder numbering.)

for every vertex $u \in S$, $u \neq r$, edge $\{u, p(u)\}$ is traversed exactly once, from $p(u)$ to $u$. This implies that after the for-loop is finished, list $\mathcal{E}$ represents a tour from $s'$ to the last vertex in $T$ such that all edges are traversed twice, except the edges between the vertices on stack $S$. These edges have to be traversed once more, from children toward the parents. This is accomplished by the while-loop in Lines 15–20. The root $r$ is added to the end of $\mathcal{E}$ by this while-loop, so that we have to remove it, in order not to store vertex $r$ one more time in the list than it is visited in the Euler tour. It remains to show the invariant of the for-loop.

We show the invariant by induction on the preorder number of $v$. If $v = 1$, then $v = r$. In this case, Lines 4 and 5 of the loop are executed. As a result, stack $S$ stores the path from $r$ to $r$. There are no vertices $v < r$, and there are no edges between vertices on $S$. Thus, the remainder of the invariant is trivially true.

If $v > 1$, Lines 7–12 are executed. In Lines 7–9, vertices are removed from the stack and their parents are appended to $\mathcal{E}$ until the top of the stack stores the parent of $v$. This is equivalent to traversing edge $\{u, p(u)\}$, for each removed vertex $u$, so that the invariant is maintained that for each vertex $u < v$, $u \notin S$, edge $\{u, p(u)\}$ is being traversed twice by the tour described by $\mathcal{E}$. After Line 9, stack $S$ represents a path from $r$ to $p(v)$, and the tour described by $\mathcal{E}$ traverses $T$ from $r$ to $p(v)$ and visits all edges between vertices on stack $S$ exactly once. Lines 11 and 12 add $v$ to $S$ and $\mathcal{E}$, so that stack $S$ now represents the path from $r$ to $v$, and edge $\{p(v), v\}$ is being traversed by the tour described by list $\mathcal{E}$.

In order to complete the proof, we need to show that for each vertex $v$, $p(v) \in S$ before the iteration for vertex $v$ is entered. In order to show that, we have to show that $p(v)$ is an ancestor of $v - 1$. If this were not true, then $p(v) < v - 1 < v$, $v$ is in the subtree of $T$ rooted at $p(v)$, and $v - 1$ is not. This contradicts the fact that a preorder numbering assigns consecutive numbers to the vertices in each subtree rooted at some vertex $x$. □

**Lemma 7.17** *List $\mathcal{T}'$, as computed by Lines 22–34 of procedure* RootTree *(Algorithm 7.9), stores the vertices of tree $T$, rooted at $s'$, sorted in preorder. Every vertex in $\mathcal{T}'$ is labelled with its parent in $T$.*

*Proof.* By definition, a preorder numbering of $T$ is a numbering of the vertices in $T$ according to the order of their first appearances in an Euler tour of $T$. Thus, we only have to show that Lines 23–34 of procedure RootTree extract the first appearance of each vertex in $\mathcal{E}'$. Also, the first appearance of each vertex $v$ in an Euler tour of $T$ is preceded by an appearance of the parent of $v$. Thus, if the algorithm extracts only the first appearance of each vertex, it also computes the parent of each vertex correctly.

A vertex is extracted from $\mathcal{E}'$ if it is not equal to the top of the stack. This is true for the first appearance of each vertex in $\mathcal{E}'$, as vertices are pushed on the stack only when they are visited. It remains to show that every vertex of $T$ is extracted exactly once. In order to do that, we show that each but the first appearance of each

vertex $v$ finds $v$ on the top of the stack, so that $v$ is not appended to $\mathcal{T}'$ again. The proof is by induction on the size of the subtree $T(v)$ of $T$ rooted at $v$.

If $|T(v)| = 1$, $v$ is a leaf, and $v$ appears only once in any Euler tour of $T$. Also, $v$ is never pushed on the stack $S$, as its successor in $\mathcal{E}'$ is $p(v)$.

If $|T(v)| > 1$, $v$ is an internal node. Let $w_1, \ldots, w_k$ ($k \leq 3$) be the children of $v$ in $T$. Then, by the induction hypothesis, each but the first appearance of $w_i$ finds $w_i$ on the top of the stack. In particular, the top of $S$ looks like $\langle w_i, v, \ldots \rangle$ when vertex $w_i$ is visited for the first time. Now the first appearance of $v$ precedes $w_1$, while every other appearance of $v$ immediately succeeds the last appearance of one of $v$'s children $w_i$. As each such last appearance of a child of $v$ is succeeded by $p(w_i) = v$, $w_i$ is removed from $S$ when visiting the last appearance of $w_i$, so that before visiting the next appearance of $v$, $v$ is on the top of stack $S$. This proves the correctness of procedure RootTree. □

**Pruning subtrees.** Having changed the order of the vertices in $T$ so that they are sorted according to a preorder numbering of $T$ starting at $s'$, we show how to compute a weight $\omega'(e)$, for every edge $e = \{x, y\} \in G$, so that $\omega'(e) = d_{G_{e^*}}(x, y)$.

For all exterior edges $e$ of $G$, let $\omega'(e) = \omega(e)$. Next we compute edge weights $\omega'(e)$, for all diagonals $e$ of $G$. To do this, we process $T$ bottom-up. For a vertex $v \neq s'$ of $T$, let $e = \{v, p(v)\}$, $e^* = \{x_e, y_e\}$, and $z \in V_v \setminus \{x_e, y_e\}$. Then let $\omega'(e^*) = \min(\omega(e^*), \omega'(\{x_e, z\}) + \omega'(\{y_e, z\}))$.

This computation takes $\mathcal{O}(\text{scan}(N))$ I/Os using the time-forward processing procedure for rooted trees. The following lemma shows that it produces the correct result.

**Lemma 7.18** *For every edge $e \in T$, $\omega'(e^*) = d_{G_e}(x_e, y_e)$.*

*Proof.* We prove this claim by induction on the size of $T_e$. If $|T_e| = 1$, then $e = \{v, p(v)\}$, where $v$ is a leaf of $T$. In this case, $G_e = \Delta(v)$, and the shortest path from $x_e$ to $y_e$ in $G_e$ is either the edge $e^* = \{x_e, y_e\}$ itself, or it is the path $P = (x_e, z, y_e)$, where $z$ is the third vertex of $\Delta(v)$. Edges $e_1 = \{x_e, z\}$ and $e_2 = \{y_e, z\}$ are exterior edges

of $G$, so that $\omega'(e_1) = \omega(e_1)$ and $\omega'(e_2) = \omega(e_2)$. Thus, $\omega'(e^*)$ is correctly computed as the minimum of the weights of edge $e^*$ and path $P$.

If $|T_e| > 1$, let $e = \{v, p(v)\}$, $w_1$ and $w_2$ be the children of $v$ in $T$, and let $e_1$ and $e_2$ be the edges $\{v, w_1\}$ and $\{v, w_2\}$. Let $e_1^* = \{x_{e_1}, y_{e_1}\}$ and $e_2^* = \{x_{e_2}, y_{e_2}\}$, where $x_{e_1} = x_e$, $y_{e_2} = y_e$, and $y_{e_1} = x_{e_2}$. If vertex $w_2$ does not exist, assume that $e_2$ is a tree-edge connecting $v$ to an artificial vertex in the outer face of $G$ and $G_{e_2} = (\{x_{e_2}, y_{e_2}\}, \{\{x_{e_2}, y_{e_2}\}\})$. Then $G_e = G_{e_1} \cup G_{e_2} \cup (\{x_e, y_e\}, e^*)$. By the induction hypothesis, $\omega'(\{x_e, y_{e_1}\}) = d_{G_{e_1}}(x_e, y_{e_1})$ and $\omega'(\{x_{e_2}, y_e\}) = d_{G_{e_2}}(x_{e_2}, y_e)$. Thus, $\omega'(e^*) = \min\{\omega(e^*), \omega'(\{x_e, y_{e_1}\}) + \omega'(\{x_{e_2}, y_e\}) = d_{G_e}(x_e, y_e)$. □

**Computing distances to the source.** In order to compute $d_G(s, v)$, for all vertices $v \in G$, tree $T$ is processed top-down, maintaining the property that after processing vertex $v \in T$, the distances $d(s, x)$ have been computed for all vertices $x \in \Delta(v)$. At the root $s'$ of $T$, $s \in \Delta(s')$. For the other two vertices $v$ and $w$ of $\Delta(s')$,

$$d_G(s, v) = \min(\omega'(\{s, v\}), \omega'(\{s, w\}) + \omega'(\{w, v\})) \text{ and}$$
$$d_G(s, w) = \min(\omega'(\{s, w\}), \omega'(\{s, v\}) + \omega'(\{v, w\})).$$

At any other vertex $v \in T$, let $e = \{v, p(v)\}$. Then $\Delta(v)$ has vertices $x_e$, $y_e$, and a third vertex $z$ on its boundary. After processing the parent of $v$, $d_G(s, x_e)$ and $d_G(s, y_e)$ are known. Let $d_G(s, z) = \min(d_G(s, x_e) + \omega'(\{x_e, z\}), d_G(s, y_e) + \omega'(\{y_e, z\}))$.

Again, this procedure takes $\mathcal{O}(\text{scan}(N))$ I/Os, using the time-forward processing procedure for rooted trees. The following lemma shows that it produces the correct result.

**Lemma 7.19** *The above procedure computes $d_G(s, v)$ correctly, for all $v \in G$.*

*Proof.* Observe that the distance $d_G(s, v)$ is computed when processing a node $x_v \in T$ such that $v \in \Delta(x_v)$, and $v \notin \Delta(p(x_v))$. We prove that $d_G(s, v)$ is computed correctly, by induction on $d_T(s', x_v)$. If $d_T(s', x_v) = 0$, then $v \in \Delta(s')$. If $s = v$, then $d_G(s, v) = 0$ is correctly computed by our algorithm. Otherwise, let the vertex set of $\Delta(s')$ be $\{s, v, x\}$, and the edges of $\Delta(s')$ be $e_1^* = \{s, v\}$, $e_2^* = \{s, x\}$, and $e_3^* = \{x, v\}$.

By Lemma 7.18, $\omega'(e_1^*) = d_{G_{e_1}}(s,v)$, $\omega'(e_2^*) = d_{G_{e_2}}(s,x)$, and $\omega'(e_3^*) = d_{G_{e_3}}(x,v)$. Thus, $d_G(s,v) = \min(d_{G_{e_1}}(s,v), d_{G_{e_2}}(s,x) + d_{G_{e_3}}(x,v)) = \min\{\omega'(e_1^*), \omega'(e_2^*) + \omega'(e_3^*))$, as computed by the algorithm.

If $d_T(s', x_v) = k > 0$, assume that the distances are computed correctly for all vertices $w$ with $d_T(s', x_w) < k$. Let $e = \{x_v, p(x_v)\}$. Then the vertex set of $\Delta(x_v)$ is $\{x_e, y_e, v\}$, and $d_T(s', x_{x_e}) < k$ and $d_T(s', x_{y_e}) < k$. Thus, the distances $d(s, x_e)$ and $d(s, y_e)$ have already been computed correctly. Let $e_1^* = \{x_e, v\}$, and $e_2^* = \{y_e, v\}$. The shortest path from $s$ to $v$ is either the concatenation of the shortest path from $s$ to $x_e$, followed by the shortest path from $x_e$ to $v$ in $G_{e_1}$, or the shortest path from $s$ to $y_e$, followed by the shortest path from $y_e$ to $v$ in $G_{e_2}$. Thus, $d_G(s,v) = \min(d_G(s,x_e) + d_{G_{e_1}}(x_e, v), d_G(s,y_e) + d_{G_{e_2}}(y_e, v)) = \min(d_G(s,x_e) + \omega'(e_1^*), d_G(s,y_e) + \omega'(e_2^*))$, as computed by the algorithm. □

**Extracting the tree.** In order to extract a shortest path tree from $s$ to all other vertices in $G$, we augment the three steps of our algorithm. The first phase, which roots tree $T$ at root $s'$, does not need to be changed. We augment the second phase as follows:

For every tree edge $e \in T$, let $e = \{v, p(v)\}$, and $z \in \Delta(v) - \{x_e, y_e\}$. Depending on whether $\omega'(e^*)$ has been computed as $\omega(e^*)$ or $\omega'(\{x_e, z\}) + \omega'(\{z, y_e\})$, the shortest path from $x_e$ to $y_e$ in $G_e$ is either edge $e^*$ or the concatenation of the shortest paths from $x_e$ to $z$ and from $z$ to $y_e$ in $G_e$. We store a flag with edge $e$ to distinguish between these two possibilities.

The third phase of the algorithm now proceeds as follows: Let $d_G(s,v) = d_G(s,x) + \omega'(\{x,v\})$, and assume that the parent of $v$ has not been computed yet. If $\{x,v\}$ is an external edge of $G$, we add edge $\{x,v\}$ to the shortest path tree and set $p(v) = x$. Otherwise, there are two possibilities. If $\omega'(\{x,v\}) = \omega(\{x,v\})$, we add edge $\{x,v\}$ to the shortest path tree, set $p(v) = x$, and inform all descendants $w$ of $x_v$ such that $v \in \Delta(w)$ that the parent of $v$ has already been computed. Otherwise, we inform the descendant $w$ of $x_v$ such that $\{x,v\} = \{w, x_v\}^*$ that $v$'s parent lies in $G_{\{w,x_v\}}$ and still needs to be computed.

The correctness of this procedure is easily verified, given that the above algorithm computes distances $d_G(s, v)$ correctly. Thus, we have shown the following theorem.

**Theorem 7.5** *Given a list $\mathcal{A}$ representing an outerplanar embedding $\hat{G}$ of an outerplanar graph $G = (V, E)$ and a weight function $\omega : E \to \mathbb{R}$ so that $G$ does not contain negative cycles, it takes $\mathcal{O}(\text{scan}(N))$ I/Os to compute a BFS-tree for $G$ or to solve the single source shortest path problem for $G$.*

*Proof.* The correctness of the above algorithm follows from the above discussion. All steps of the algorithm, except the first one, process a tree $T$ of size $\mathcal{O}(N)$ using the linear-I/O time-forward processing procedure for rooted trees. Thus, they take $\mathcal{O}(\text{scan}(N))$ I/Os. As for the first step, Lines 1–21 of procedure RootTree read list $\mathcal{T}$ and produce list $\mathcal{E}$. The size of list $\mathcal{E}$ is bounded by the number of stack operations performed because at most one element per stack operation is added to $\mathcal{E}$. The number of Pop operations is bounded by the number of Push operations, which in turn is bounded by $|\mathcal{T}|$, as each element of $\mathcal{T}$ is pushed on the stack at most once. Thus, $|\mathcal{E}| = \mathcal{O}(N)$, and the algorithm performs $\mathcal{O}(N)$ stack operations. Hence, Lines 1–21 take $\mathcal{O}(\text{scan}(N))$ I/Os. Given that $|\mathcal{E}| = \mathcal{O}(N)$, $|\mathcal{E}'| = \mathcal{O}(N)$, and Line 22 takes $\mathcal{O}(\text{scan}(N))$ I/Os. In Lines 23–34, list $\mathcal{E}'$ is being scanned, and list $\mathcal{T}'$ is written. Every element in $\mathcal{E}'$ causes at most one element to be appended to $\mathcal{T}'$ and at most one element to be pushed on the stack. Thus, $|\mathcal{T}'| = \mathcal{O}(N)$, and $\mathcal{O}(N)$ stack operations are performed. Hence, Lines 23–34 also take $\mathcal{O}(\text{scan}(N))$ I/Os. This shows that the whole algorithm takes $\mathcal{O}(\text{scan}(N))$ I/Os to solve the single source shortest path problem. In order to compute a BFS-tree, we give all edges in $G$ unit weight. □

## 7.5 Lower Bounds

In this section, we address the question whether the algorithms presented in this chapter are optimal. Note that all the problems in this chapter require $\Omega(\text{scan}(N))$ I/Os. Given an outerplanar embedding, we present optimal linear-I/O algorithm for these problems. However, if no outerplanar embedding of the graph is given, our algorithms

spend $\mathcal{O}(\text{perm}(N))$ I/Os to solve any of the problems discussed in this chapter, as they have to obtain an outerplanar embedding of the graph first. Now the question is whether the embedding step can be avoided, in order to obtain true linear-I/O algorithms for BFS, DFS, SSSP, triangulation, or outerplanar separators.

Before being able to show a lower bound for any of these problems, we have to define exactly how the output of an algorithm solving the problem is to be represented. For most of the problems discussed here, such a representation of the output is far from well-defined. For instance, a graph is said to be embedded if the circular order of the edges incident to each vertex is given. How this order is to be represented is left to the particular algorithm. It may be represented as a numbering or sorting of the edges clockwise around the vertex, or by having each edge store pointers to its successors clockwise around each of its endpoints. The output of a BFS-algorithm may be a labelling of each vertex with its distance to the root of the BFS-tree, or just a representation of the BFS-tree by computing for each vertex, its parent in the BFS-tree.

Even though we believe that, if no embedding of the graph is given as part of the input, $\Omega(\text{perm}(N))$ is a lower bound on the number of I/Os required to compute an embedding, BFS-tree, DFS-tree, or shortest path tree of an outerplanar graph $G$, independent of which representation of the output is chosen, we are only able to prove such a lower bound, if we place certain restrictions on the output representation of the respective algorithm. These restrictions are satisfied by our algorithms. We discuss these restrictions next.

For each vertex $v \in G$, let $A(v)$ be its adjacency list. Then we require that an algorithm computing an outerplanar embedding of $G$ either numbers the vertices in $A(v)$ from 1 to $|A(v)|$ clockwise or counterclockwise around $v$, or produces a representation that can be transformed into this representation of the embedding in $o(\text{perm}(N))$ I/Os. Our algorithm produces lists $A(v)$ sorted counterclockwise around $v$. Such a representation can be transformed into a numbering of the vertices in $A(v)$ in a single scan of lists $A(v)$, $v \in G$.

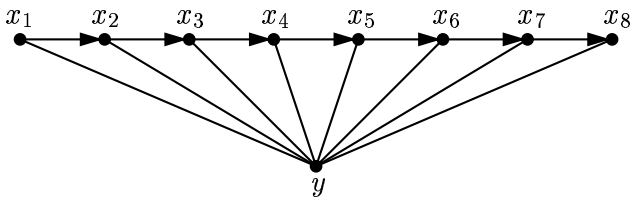



**Figure 7.6**
The proof of the lower bound for outerplanar embedding.

A BFS, DFS, or SSSP-algorithm is required to label each vertex $v \in G$ with the length of the shortest path in $T$ from the root of $T$ to $v$, or produce an output that allows the computation of such distance labels in $o(\text{perm}(N))$ I/Os. Our BFS and SSSP-algorithms compute such distance labels. The DFS-algorithm represents the computed spanning tree $T$ by a list $\mathcal{T}$ storing its vertices in preorder. Distance labels of the vertices in $G$ can now easily be computed in a linear number of I/Os by processing $T$ from the root toward the leaves.

Our lower bound proofs use a linear-I/O reduction from list-ranking to the problem whose lower bound we want to show. This implies that the problem requires $\Omega(\text{perm}(N))$ I/Os to be solved, since list-ranking has an $\Omega(\text{perm}(N))$ I/O lower bound [43].

**Lemma 7.20** *Given a list $L$ containing $N$ elements, it takes $\mathcal{O}(\text{scan}(N))$ I/Os to construct an outerplanar graph $G_L$ of size $\mathcal{O}(N)$ from $L$ and extract a ranking $\delta$ of $L$ from an outerplanar embedding of $G_L$.*

*Proof.* We define graph $G_L = (V_L, E_L)$ as follows: The vertex set of $G_L$ is defined as the set $V_L = \{x_1, \ldots, x_N, y\}$. The edge set of $G_L$ is defined as $E_L = \{\{x_i, x_{i+1}\} : 1 \leq i < N\} \cup \{\{y, x_i\} : 1 \leq i \leq N\}$. Graph $G_L$ can easily be constructed from list $L$ in $\mathcal{O}(\text{scan}(N))$ I/Os. Figure 7.6 shows an outerplanar embedding of $G_L$. By Lemma 7.2, this is the only possible embedding of $G_L$, except for flipping the whole graph. Thus, an algorithm numbering the vertices in $A(y)$ clockwise around $y$ produces a labelling $\delta' : L \to \mathbb{N}$ such that either $\delta'(x_i) = (i + c) \bmod N$, where $c \in \mathbb{N}$ is a constant, or $\delta'(x_i) = (c - i) \bmod N$. It is straightforward to decide which of the two cases applies

and determine constant $c$, based on the labels $\delta'(x_1)$ and $\delta'(x_N)$. Then a single scan of the vertices in $L$ suffices to transform the labelling $\delta'$ into the desired labelling $\delta$. □

The following simple reduction shows that any algorithm computing a BFS, DFS, or shortest path tree for an outerplanar graph $G$ requires $\Omega(\text{perm}(N))$ I/Os, even if the choice of the root vertex is left to the algorithm.

**Lemma 7.21** *Given a list $L$ containing $N$ elements, it takes $\mathcal{O}(\text{scan}(N))$ I/Os to extract a ranking $\delta$ of $L$ from a BFS-tree, DFS-tree, or shortest path tree of $L$, when viewed as an undirected graph $G_L$.*

*Proof.* We represent list $L$ as the graph $G_L = (V_L, E_L)$, $V_L = \{x_1, \ldots, x_N\}$, $E_L = \{\{x_i, x_{i+1}\} : 1 \leq i < N\}$. For the SSSP problem, we assign unit weights to the edges of $G_L$. Given that the algorithm chooses vertex $x_k$, $1 \leq k \leq N$, as the root of the spanning tree it computes, the vertices of $G_L$ are labelled with their distances $\delta'(x_i)$ from $x_k$. In particular, $\delta'(x_i) = k - i$ if $1 \leq i \leq k$, and $\delta'(x_i) = i - k$ if $k < i \leq N$. The distance label $\delta'(x_1)$ is sufficient to determine index $k$. Then it takes a single scan of the vertices in $L$ to transform the labelling $\delta'$ into the desired labelling $\delta$. □

Together with the $\Omega(\text{perm}(N))$ I/O lower bound for list-ranking, Lemmas 7.20 and 7.21 imply the following result.

**Theorem 7.6** *Given an outerplanar graph $G = (V, E)$ with $N$ vertices, represented as vertex and edge lists, it takes $\Omega(\text{perm}(N))$ I/Os to compute an outerplanar embedding, DFS-tree, BFS-tree, or solve the single source shortest path problem for $G$.*

# Chapter 8

# Planar Separators

In this chapter, we present an I/O-efficient algorithm for computing a small $\varepsilon$-separator of an unweighted planar graph. The algorithm is superior to all existing I/O-efficient algorithms for this problem, as the latter require a BFS-tree and a planar embedding of the graph to be given as part of the input, while our algorithm does not require this extra information. In fact, we use this separator algorithm in Chapters 9 and 10 to compute a planar embedding and a BFS-tree of a planar graph I/O-efficiently.

In Section 8.1, we present some preliminary results that are used by our algorithm. In Sections 8.2 through 8.5, we present the algorithm.

## 8.1 Preliminaries

### 8.1.1 Separator Theorems

In our separator algorithm, we make use of the following two results. The first result provides us with a linear-time separator algorithm which guarantees a bound on the total size of the separator and the size of each particular subgraph, but does not give a guarantee on the boundary size of each particular subgraph. The second algorithm

is by a $\log N$ factor slower, but guarantees that the boundary of each subgraph is small.

We use the first result to compute an intermediate partition in an I/O-efficient manner. The second result is then applied to each subgraph of this partition. Since these subgraphs are small, we can apply the second result in internal memory, so that the extra $\log N$ factor does not increase the I/O-complexity of our algorithm.

**Theorem 8.1 (Aleksandrov and Djidjev [8])** *Given a planar graph $G = (V, E)$ and a real constant $0 < \varepsilon < 1$, an $\varepsilon$-separator of size $\mathcal{O}\left(\sqrt{N/\varepsilon}\right)$ can be computed in linear time.*

**Theorem 8.2 (Frederickson [73])** *Given a planar graph $G = (V, E)$, a normal $h$-partition $\mathcal{P} = (S, \{G_1, \ldots, G_k\})$ of $G$, and a constant $c > 0$, a $c$-proper $h$-partition $\mathcal{P}' = (S', \{G'_1, \ldots, G'_l\})$ of $G$ with $S' \supseteq S$ can be computed in $\mathcal{O}(N \log N)$ time and linear space.*

### 8.1.2 Bipartite Planar Graphs

We will apply the following lemma and its corollary to prove that a greedy graph contraction procedure used in our algorithm produces a graph that is guaranteed to be small.

**Lemma 8.1** *Let $G = (V_1, V_2, E)$ be a simple connected bipartite planar graph such that every vertex in $V_2$ has degree at least three. Then $|V_2| \leq 2|V_1|$.*

*Proof.* Consider a planar embedding $\hat{G}$ of $G$. As $G$ is bipartite, every face of $\hat{G}$ has size at least 4. Thus, $|F| \leq |E|/2$. By Euler's formula, $|V| + |F| - |E| = 2$. That is,

$$\begin{aligned} 2 &= |V| + |F| - |E| \\ &\leq |V| - |E|/2 \\ |E| &\leq 2|V|. \end{aligned}$$

On the other hand, $|E| \geq 3|V_2|$, so that

$$\begin{aligned} 3|V_2| &\leq 2|V| \\ &= 2(|V_1| + |V_2|) \\ |V_2| &\leq 2|V_1|. \end{aligned}$$

□

**Corollary 8.1** *Let $G = (V_1, V_2, E)$ be a bipartite planar graph. Let the vertices in $V_2$ be partitioned into equivalence classes $C_1, \ldots, C_q$, where two vertices $v$ and $w$ are equivalent if they have degree at most two and are adjacent to the same set of vertices in $V_1$. Then $q \leq 6|V_1|$.*

*Proof.* Every vertex $v \in V_1$ defines one class $C_v$ such that the vertices in $C_v$ are adjacent only to $v$. There are at most $|V_1|$ such classes $C_v$. Now consider a pair $\{v, w\}$ of vertices in $V_1$. Let $C_{\{v,w\}}$ be the set of vertices in $V_2$ adjacent only to $v$ and $w$. Then we choose one representative $r_{\{v,w\}} \in C_{\{v,w\}}$ for each such class of vertices. Let $H_1$ be the bipartite subgraph of $G$ induced by all edges incident to these representatives $r_{\{v,w\}}$. As $H_1$ is a subgraph of $G$, $H_1$ is planar. We remove representatives $r_{\{v,w\}}$ from $H_1$ and replace edges $\{v, r_{\{v,w\}}\}$ and $\{r_{\{v,w\}}, w\}$ by a single edge $\{v, w\}$, for every class $C_{\{v,w\}}$. As every representative $r_{\{v,w\}}$ has degree two, the resulting graph $H_2$ is a planar graph whose vertex set is a subset of $V_1$, and whose edges are in one-to-one correspondence to classes $C_{\{v,w\}}$. Thus, by Euler's formula, there can be at most $3|V_1|$ such classes $C_{\{v,w\}}$. Finally, let $H_3$ be the subgraph of $G$ induced by all edges incident to vertices of degree at least three in $V_2$. Graph $H_3$ is a bipartite planar graph $H_3 = (V_1', V_2', E')$ with $V_1' \subseteq V_1$, $V_2' \subseteq V_2$, and $E' \subseteq E$. All vertices in $V_2'$ have degree at least three, so that $|V_2'| \leq 2|V_1'| \leq 2|V_1|$, by Lemma 8.1. Each vertex in $V_2'$ is in its own class, so that there are at most $6|V_1|$ classes in total. □

## 8.2 Overview of the Algorithm

We use Algorithm 8.1 to compute a proper $h$-partition $\mathcal{P} = (S, \mathcal{R})$ of $G$. The algorithm takes $\mathcal{O}(\text{sort}(N))$ I/Os and uses linear space, provided that $M \geq 56h\log^2(DB)$.

The loop in Lines 3–5 of Algorithm 8.1 constructs a hierarchy of $\log_2(DB)$ graphs $G_0, \ldots, G_r$ such that $G = G_0$, $|G_{i+1}| \leq \frac{1}{2}|G_i|$, and every vertex in $G_{i+1}$ represents a small subgraph of $G_i$ and thus a subgraph of $G$. Given this hierarchy, Line 6 computes a small separator $S_r$ whose removal partitions $G_r$ into relatively coarse subgraphs. The number of vertices in $G$ corresponding to the vertices in $S_r$ is small. Lines 7–9 undo the contraction steps used to compute graphs $G_1, \ldots, G_r$ one by one. Every iteration starts with a partition of the current graph $G_i$ induced by the partition of graph $G_{i+1}$ computed in the previous iteration, and then refines this partition to obtain a suitable partition of graph $G_i$. At the end of the last iteration, a separator of size $\mathcal{O}\left(N/\sqrt{h}\right)$ is obtained whose removal partitions $G$ into subgraphs of size at most $h\log^2(DB)$. Given such a partition, Line 10 loads each subgraph into internal memory and partitions it into subgraphs of size at most $h$ and boundary size at most $\sqrt{h}$. This can be done using Theorems 8.1 and 8.2 and introduces at most $\mathcal{O}\left(N/\sqrt{h}\right)$ additional separator vertices. Thus, Line 10 produces a separator of size $\mathcal{O}\left(N/\sqrt{h}\right)$ whose removal partitions $G$ into subgraphs of size at most $h$ and boundary size at most $\sqrt{h}$. In order to ensure that the resulting partition $\mathcal{P}$ contains only $\mathcal{O}(N/h)$ subgraphs, the connected components in the partition are grouped appropriately before performing the computation of Line 10.

Our algorithm can be divided into two phases. The first phase (Lines 3–9) is only concerned with generating a partition into subgraphs of size at most $h\log^2(DB)$ while keeping the total separator size small. The boundary size of each subgraph may be large at the end of this phase. The second phase (Lines 10) partitions each subgraph further, so that every subgraph has size at most $h$ and boundary size at most $\sqrt{h}$. Next we describe these phases in detail.

**Procedure** SEPARATOR

**Input:** A planar graph $G = (V, E)$ and an integer $h > 0$.
**Output:** A proper $h$-partition $\mathcal{P} = (S, \mathcal{R})$ of $G$.

1: $G_0 \leftarrow G$
2: $r \leftarrow \lceil \log(DB) \rceil - 1$
3: **for** $i = 1, \ldots, r$ **do**
4: Compute a graph $G_i$ with the following properties from $G_{i-1}$:

(i) Graph $G_i$ is planar,

(ii) $|G_i| \leq \frac{28N}{2^{i+1}}$,

(iii) Every vertex in $G_i$ represents a subgraph of constant size in $G_{i-1}$, and

(iv) Every vertex in $G_i$ represents at most $2^{i+1}$ vertices in $G$.

5: **end for**
6: Apply Theorem 8.1 to compute a separator $S_r$ of size $\mathcal{O}\left(N/\left(\sqrt{h}\log(DB)\right)\right)$ for $G_r$ whose removal partitions $G_r$ into connected components of size at most $h\log^2(DB)$.
7: **for** $i = r-1, \ldots, 0$ **do**
8: Compute a separator $S_i$ for $G_i$. Separator $S_i$ consists of two sets $S_i'$ and $S_i''$. $S_i'$ is the set of vertices in $G_i$ represented by the vertices in $S_{i+1}$. $S_i''$ is the set of separator vertices introduced in order to partition the connected components of $G_i - S_i'$ into subgraphs of size at most $h\log^2(DB)$.
9: **end for**
10: Compute partition $\mathcal{P} = (S, \mathcal{R})$ by partitioning the connected components of $G - S_0$ into $\mathcal{O}(N/h)$ subgraphs of size at most $h$ and boundary size at most $\sqrt{h}$, and adding the required separator vertices to $S_0$.

**Algorithm 8.1**
Computing a proper $h$-partition for an unweighted planar graph.

## 8.3 The Graph Hierarchy

The first step of our algorithm is to compute the sequence of graphs $G_0, \ldots, G_r$. We first show how to construct graph $G_{i+1}$ from graph $G_i$, for $0 \leq i < r$, and then prove a number of useful properties of graphs $G_0, \ldots, G_r$.

Given a graph $G_i$ in the hierarchy, we assign a *weight* $\omega(v)$ to every vertex $v \in G_i$. This weight equals the number of vertices in $G$ represented by $v$. That is, $\omega(v) = 1$, for

every vertex $v \in G_0$, as $G_0 = G$. For every vertex $v \in G_i$, its *size* $\sigma(v)$ is the number of vertices in $G_{i-1}$ represented by $v$. We define a series of weight thresholds $\rho_i = 2^{i+1}$, for $0 \leq i \leq r$. Given $G_i$, we define a graph $G'_{i+1} = G_i$ with $\sigma(v) = 1$, for all $v \in G'_{i+1}$. The weight of a vertex in $G'_{i+1}$ is the same as its weight in $G_i$. Now we inspect the edges of $G'_{i+1}$ in an arbitrary order. As long as there is an edge $\{v, w\} \in G'_{i+1}$ such that $\omega(v) + \omega(w) \leq \rho_{i+1}$ and $\sigma(v) + \sigma(w) \leq 56$, we contract edge $\{v, w\}$ and repeat. Let $G''_{i+1}$ be the resulting graph which does not allow any further edge contractions.

We call a vertex $v$ in $G''_{i+1}$ *heavy* if $\omega(v) \geq \rho_{i+1}/2$ or $\sigma(v) \geq 28$, and *light* otherwise. Note that for every edge $\{v, w\} \in G''_{i+1}$, either $\omega(v)+\omega(w) > \rho_{i+1}$ or $\sigma(v)+\sigma(w) > 56$, so that at least one of $v$ and $w$ is heavy. Thus, no two light vertices are adjacent. Now we partition the light vertices of degree at most two in $G''_{i+1}$ into classes $C_1, \ldots, C_q$ such that the vertices in each class have the same set of (heavy) neighbors. Each class $C_j$ is partitioned into a minimal number of subclasses $C_{j,1}, \ldots, C_{j,k_j}$ so that the total weight of each subclass $C_{j,l}$, $1 \leq l \leq k_j$, is at most $\rho_{i+1}$ and its total size is at most 56. For $1 \leq l < k_j$, either the total weight of $C_{j,l}$ is at least $\rho_{i+1}/2$ or its total size is at least 28. The last class $C_{j,k_j}$ may have weight less than $\rho_{i+1}/2$ and size less than 28.

Now graph $G_{i+1}$ is defined as follows: The vertex set of $G_{i+1}$ consists of all heavy vertices of $G''_{i+1}$, all light vertices of degree at least three, as well as one vertex $v_{j,l}$, for each class $C_{j,l}$ of light vertices of degree at most two. For every heavy vertex $v \in G''_{i+1}$, the weight of $v$ in $G_{i+1}$ is the same as in $G''_{i+1}$. The same is true for all light vertices of degree at least three. For every vertex $v_{j,l}$, let $\omega(v_{j,l}) = \sum_{v \in C_{j,l}} \omega(v)$. There is an edge between two vertices $v$ and $w$, where $v$ and $w$ are heavy or of degree at least three, if there is an edge between $v$ and $w$ in $G''_{i+1}$. There is an edge $\{v, v_{j,l}\} \in G_{i+1}$, where $v$ is a heavy vertex and $v_{j,l}$ corresponds to class $C_{j,l}$, if the vertices in $C_{j,l}$ are adjacent to $v$ in $G''_{i+1}$. Next we prove some properties of graphs $G_0, \ldots, G_r$ which are crucial to guarantee an upper bound on the size of the separator computed by Algorithm 8.1.

**Lemma 8.2** *Let $G$ be a planar graph, and let $G = G_0, \ldots, G_r$ be the graphs as defined above. Every graph $G_i$, $0 \leq i \leq r$, has the following properties:*

*(i) Graph $G_i$ is planar,*

*(ii) $\omega(v) \leq \rho_i$, for all vertices $v \in G_i$,*

*(iii) Every vertex $v \in G_i$ corresponds to at most 56 vertices in $G_{i-1}$, for $i > 0$, and*

*(iv) $|G_i| \leq 28N/\rho_i$.*

*Proof.* To prove Property (i), we inductively show that each graph $G_i$, $0 \leq i \leq r$, has a planar embedding $\hat{G}_i$. For $i = 0$, $G_0 = G$, so that an embedding $\hat{G}_0$ of $G_0$ exists by the planarity of $G$. So assume that we are given an embedding $\hat{G}_{i-1}$ of graph $G_{i-1}$. We show how to obtain a planar embedding $\hat{G}_i$ of $G_i$. First we construct a planar embedding of $G_i''$. Such an embedding is easily obtained from $\hat{G}_{i-1}$, as $G_i''$ is constructed from $G_{i-1}$ using a series of edge contractions. The adjacencies between heavy vertices and light vertices of degree at least three do not change from $G_i''$ to $G_i$. For a class $C_{i,l}$, vertex $v_{i,l}$ is adjacent to the same heavy vertices in $G_i$ as all members of $C_{i,l}$ in $G_i''$. We rename an arbitrary member of $C_{i,l}$ to $v_{i,l}$ and remove all other members of $C_{i,l}$ from $\hat{G}_i''$. The result is the desired embedding $\hat{G}_i$ of $G_i$.

Property (ii) is easy to show by induction. In particular, $\omega(v) = 1 \leq \rho_0$, for all vertices $v \in G_0$. Given that all vertices in $G_i$ have weight at most $\rho_i$, the weight of a vertex in $G_{i+1}$ can exceed $\rho_{i+1}$ only by merging two or more vertices into a single vertex. But two vertices are merged only if their total weight does not exceed $\rho_{i+1}$. Property (iii) can be shown in a similar manner.

It remains to show Property (iv). For graph $G_0$, Property (iv) holds because $|G_0| = |G| = N$. To prove the claim for graphs $G_1, \ldots, G_r$, let $h_i$ be the number of heavy vertices in $G_i$. It follows from the above construction and Corollary 8.1 that $|G_i| \leq 7h_i$. Thus, Property (iv) follows if we can show that $h_i \leq 4N/\rho_i$.

We prove this claim by induction. The heavy vertices can be partitioned into two categories. Heavy vertices of type I are heavy because their weight is at least $\rho_i/2$. Type-II vertices are heavy because their size is at least 28. In general, graph $G_i$ contains at most $2N/\rho_i$ type-I vertices and at most $|G_{i-1}|/28$ type-II vertices. That is $h_i \leq \frac{2N}{\rho_i} + \frac{|G_{i-1}|}{28}$.

For $i = 1$, we obtain $h_1 \leq \frac{2N}{\rho_1} + \frac{N}{28} < \frac{4N}{\rho_1}$ because $\rho_1 = 4$. For $i > 1$, we obtain

$$h_i \leq \frac{2N}{\rho_i} + \frac{|G_{i-1}|}{28} \tag{8.1}$$

$$\leq \frac{2N}{\rho_i} + \frac{h_{i-1}}{4} \tag{8.2}$$

$$\leq \frac{2N}{\rho_i} + \frac{N}{\rho_{i-1}} \tag{8.3}$$

$$= \frac{4N}{\rho_i}. \tag{8.4}$$

Line (8.2) follows from Line (8.1) because $|G_{i-1}| \leq 7h_{i-1}$. Line (8.3) follows from Line (8.2) by the induction hypothesis. Line (8.4) follows from Line (8.3) because $\rho_i = 2\rho_{i-1}$. □

Next we show how to compute graphs $G_0, \ldots, G_r$ I/O-efficiently. Graph $G_0$ is already given. So assume that graphs $G_0, \ldots, G_{i-1}$ have been computed. We call an edge in $G_{i-1}$ *contractible* if $\rho(v) + \rho(w) \leq \rho_i$ and $\sigma(v) + \sigma(w) \leq 56$. The contractible subgraph $H = (W, F)$ of $G_{i-1}$ is the graph $H = G[F]$ induced by the set $F$ of contractible edges in $G$. Graph $G_i$ is computed using Algorithm 8.2. Line 1 of the algorithm applies Algorithm 8.4 to contract as many contractible edges in $G$ as possible. This corresponds to the construction of graph $G_i''$ from graph $G_{i-1}$. Line 2 applies Algorithm 8.3 to merge light vertices of degree at most two which are adjacent to the same set of neighbors until no two such vertices remain. This corresponds to the construction of graph $G_i$ from graph $G_i''$.

**Lemma 8.3** *Algorithm 8.2 performs $\mathcal{O}(\text{sort}(|G_{i-1}|))$ I/Os and uses linear space to compute graph $G_i$ from graph $G_{i-1}$.*

*Proof.* Algorithm 8.2 strictly follows the definition of $G_i$. By Lemmas 8.4 and 8.5, procedures ContractEdges and MergeLowDegree perform the two phases of the construction of $G_i$ correctly. Hence, Algorithm 8.2 constructs graph $G_i$ correctly. Line 1 of the algorithm takes $\mathcal{O}(\text{sort}(|G_{i-1}|))$ I/Os and linear space, by Lemma 8.6. Line 2 takes $\mathcal{O}(\text{sort}(|G_{i-1}|))$ I/Os and linear space, by Lemma 8.4. □

**Procedure** COMPRESS

**Input:** Graph $G_{i-1} = (V_{i-1}, E_{i-1})$ in the graph hierarchy.
**Output:** Graph $G_i = (V_i, E_i)$ in the graph hierarchy.

1: $G_i'' \leftarrow$ CONTRACTEDGES($G_{i-1}$)
2: $G_i \leftarrow$ MERGELOWDEGREE($G_i''$)

**Algorithm 8.2**
Computing graph $G_i$ from graph $G_{i-1}$.

**Lemma 8.4** *Given a graph $G = (V, E)$ whose vertices are labelled as "heavy" or "light" so that no two light vertices are adjacent, Algorithm 8.3 computes a graph $G' = (V', E')$ so that no two light vertices of degree at most two in $G'$ are adjacent to the same set of heavy vertices. The algorithm takes $\mathcal{O}(\text{sort}(|G|))$ I/Os and uses linear space.*

*Proof.* As the loop in Lines 2–5 merges light vertices of degree at most two until no two such vertices with the same set of neighbors remain, Algorithm 8.3 computes graph $G'$ correctly.

Line 1 can be realized by extracting all vertices of degree at most two and sorting them so that vertices with the same set of neighbors are stored consecutively. This produces a partition of the set of vertices of degree at most two into the desired equivalence classes. Lines 2–5 scan the resulting list of vertices and greedily merge consecutive vertices in the same equivalence class. Thus, Algorithm 8.3 takes $\mathcal{O}(\text{sort}(|G|))$ I/Os. □

It remains to provide the algorithm for contracting contractible edges, and prove that the algorithm takes $\mathcal{O}(\text{sort}(|G|))$ I/Os, where $G$ is the input graph of the algorithm. We use Algorithm 8.4 to contract contractible edges. The algorithm iterates the following two steps in Lines 5 and 6: First the edges in a maximal matching of the contractible subgraph $H$ of $G$ are contracted. Every vertex $v$ produced by such an edge contraction represents two vertices in $H$. We call vertex $v$ *matched*. Then all edges incident to unmatched vertices are contracted to guarantee that no unmatched

**Procedure** MERGELOWDEGREE

**Input:** A graph $G = (V, E)$ and a predicate ISHEAVY($v$) which returns true if vertex $v$ is heavy. For every edge $\{v, w\} \in E$, at least one of ISHEAVY($v$) and ISHEAVY($w$) is true.

**Output:** A graph $G' = (V', E')$ that contains no two light vertices of degree at most two which are adjacent to the same set of heavy vertices.

1: Partition the light vertices of degree at most two in $G$ into equivalence classes, where two vertices are equivalent if they have the same set of neighbors.
2: **while** there are two equivalent vertices $v$ and $w$ such that ISHEAVY($v$) = **false** and ISHEAVY($w$) = **false do**
3: Remove $w$ and its incident edges from $G$.
4: Adjust all weights of $v$ so that they represent the sums of the respective weights of $v$ and $w$.
5: **end while**
6: Let $G' = (V', E')$ be the resulting graph.

**Algorithm 8.3**
Merging light vertices of low degree.

vertex has an incident edge which is contractible. This guarantees that the vertex set of the contractible subgraph in the next iteration contains only matched vertices. Thus, the size of the contractible subgraph reduces by a factor of at least two from one iteration to the next, so that the sizes of the graphs processed in Lines 4–9 are geometrically decreasing. The next two lemmas show the correctness and I/O-complexity of Algorithm 8.4.

**Lemma 8.5** *The graph $G'$ produced by Algorithm 8.4 can be obtained from the input graph $G$ by contracting contractible edges. No edge in $G'$ is contractible.*

*Proof.* Procedures CONTRACTMATCHING and CONTRACTBIPARTITE produce their output graphs from their input graphs using edge contractions. By Lemmas 8.7 and 8.8, they contract only contractible edges. Thus, graph $G'$ can be obtained from graph $G$ by contracting contractible edges. Graph $G'$ contains no contractible edges because the loop in Lines 4–9 is repeated until no such edges remain. □

**Procedure** ContractEdges

**Input:** A planar graph $G = (V, E)$ and an operator IsContractible($e$) which returns **true** if edge $e \in E$ is contractible and **false** otherwise.

**Output:** A planar graph $G' = (V', E')$ obtained by contracting contractible edges in $G$ until no contractible edges remain.

1: Extract the contractible subgraph $H$ of $G$.
2: $F \leftarrow (V(G), \emptyset)$
3: {Forest $F$ represents a "contraction history". In particular, node $v$ is the parent of node $w$ if vertex $w$ has been contracted into vertex $v$.}
4: **while** $H$ is not empty **do**
5: $(H_1, F_1) \leftarrow$ ContractMatching($H, F$)
6: $(H_2, F_2) \leftarrow$ ContractBipartite($H_1, F_1$, IsContractible)
7: $F \leftarrow F_2$
8: Extract the contractible subgraph $H$ of $H_2$.
9: **end while**
10: Construct $G'$ from $G$: Replace every vertex in $G$ with the root of its tree in $F$ and remove duplicates. Replace the endpoints of all edges in $G$ with the roots of their trees in $F$ and remove duplicate edges.

**Algorithm 8.4**
A generic graph compression algorithm.

**Lemma 8.6** *Algorithm 8.4 takes $\mathcal{O}(\text{sort}(|G|))$ I/Os and uses linear space.*

*Proof.* The extraction of the contractible subgraph $H$ of $G$ in Line 1 of Algorithm 8.4 takes $\mathcal{O}(\text{sort}(|G|))$ I/Os: We scan the edge list of $G$ and extract all contractible edges. Then we produce a list $L$ of the endpoints of contractible edges and apply procedure DuplicateRemoval to obtain the vertex set of $H$.

Let $H^{(1)}, H^{(2)}, \ldots$ be snapshots of graph $H$ at the beginning of each iteration in Line 4. By Lemmas 8.7 and 8.8, the calls to procedures ContractMatching and ContractBipartite in Lines 5 and 6 take $\mathcal{O}\left(\text{sort}\left(\left|H^{(i)}\right|\right)\right)$ I/Os. By Lemma 8.9, $\left|H^{(i+1)}\right| \leq \frac{1}{2}\left|H^{(i)}\right|$. Thus, the total I/O-complexity of the loop in Lines 4–9 is $\mathcal{O}\left(\text{sort}\left(\left|H^{(1)}\right|\right)\right) = \mathcal{O}(\text{sort}(|G|))$. The root of the tree in $F$ containing every vertex $v \in G$ can be found in $\mathcal{O}(\text{sort}(|G|))$ I/Os by computing the connected components of $F$ [43] and then applying the Euler tour technique and list-ranking to each of the

**Procedure** CONTRACTMATCHING

**Input:** A graph $H = (V, E)$ whose edges are contractible and a forest $F$ of rooted trees whose roots are the vertices in $V$.

**Output:** The graph $H' = (V', E')$ obtained from $H$ by contracting the edges in a maximal matching of $H$ and a supergraph $F'$ of $F$ which has been augmented with edges to reflect the performed contractions.

1: Label every vertex of $H$ as "unmatched".
2: Compute a maximal matching $\mathcal{M} \subseteq E$ of $H$.
3: **for** every edge $e \in \mathcal{M}$ **do**
4: Contract $e = \{v, w\}$. Label vertex $v$ as "matched" and make $v$ the parent of $w$ in $F$.
5: **end for**
6: Let $H' = (V', E')$ and $F'$ be the resulting graphs.

**Algorithm 8.5**
Contracting the edges in a maximal matching of the contractible subgraph of $G$.

trees of $F$. Finally, we apply procedures COPYVERTEXLABELS and DUPLICATEREMOVAL to replace every edge endpoint $v$ by the root of the tree containing $v$ and remove duplicates from the resulting vertex and edge sets. This takes $\mathcal{O}(\text{sort}(|G|))$ I/Os. □

Algorithms 8.5, 8.6, and 8.7 provide the procedures for contracting a maximal matching and the edges incident to unmatched vertices in Lines 5 and 6 of Algorithm 8.4. The following lemmas show that these procedures take $\mathcal{O}(\text{sort}(|H|))$ I/Os, where $H$ is the input graph of the respective procedure. Lemma 8.9 shows that the number of vertices in the contractible subgraph of $G$ reduces by a factor of two in each iteration of the loop in Lines 4–9 of Algorithm 8.4.

**Lemma 8.7** *Algorithm 8.5 takes $\mathcal{O}(\text{sort}(|H|))$ I/Os to contract the edges in a maximal matching of $H$.*

*Proof.* Line 1 of the algorithm can be realized in a single scan of the vertex set of $H$. Line 2 takes $\mathcal{O}(\text{sort}(|H|))$ I/Os, by Theorem 6.3. Given the set of edges in $\mathcal{M}$, directed edges of the form $(w, p(w) = v)$ can be added to forest $F$ in a single scan of set $\mathcal{M}$,

**Procedure** ContractBipartite

**Input:** A graph $H = (V, E)$ whose vertices are labelled as "matched" or "unmatched", a forest $F$ of rooted trees whose roots are the vertices in $V$, and an operator IsContractible($e$) which returns **true** if and only if edge $e \in E$ is contractible.

**Output:** A graph $H' = (V', E')$ obtained from $H$ by contracting contractible edges incident to unmatched vertices until no contractible edges incident to unmatched vertices remain. A supergraph $F'$ of $F$ which has been augmented with edges to reflect the performed contractions.

```
 1: Let V_u be the set of unmatched vertices and V_m be the set of matched vertices of H.
 2: Number the vertices in V_u and V_m in their order of appearance.
 3: for all vertices v ∈ V_m do
 4:   for all vertices w ∈ V_u adjacent to v do
 5:     Let v_1 < v_2 < ··· < v_l be the vertices adjacent to w, and let v = v_j.
 6:     if j < l then
 7:       Store a pointer p_v(w) = v_{j+1} with the copy of w in the adjacency list of v.
 8:     else
 9:       Mark v as the largest vertex adjacent to w.
10:     end if
11:     if j = 1 then
12:       Mark v as the smallest vertex adjacent to w.
13:     end if
14:   end for
15: end for
```

**Algorithm 8.6**
Contracting contractible edges incident to unmatched vertices.

which takes $\mathcal{O}(\text{scan}(|H|))$ I/Os. The compression of these edges in $H$ can be achieved using procedure ContractGraph. This takes $\mathcal{O}(\text{sort}(|H|))$ I/Os. □

**Lemma 8.8** *Procedure* ContractBipartite *(Algorithms 8.6 and 8.7) ensures that there is no unmatched vertex in $H$ which has an incident edge that is contractible. The I/O-complexity of procedure* ContractBipartite *is $\mathcal{O}(\text{sort}(|H|))$.*

*Proof.* The numbering of the vertices in $V_u$ and $V_m$ in Line 2 of the algorithm can easily be carried out in $\mathcal{O}(\text{scan}(|H|))$ I/Os. The computation of Lines 3–15 can be realized in $\mathcal{O}(\text{sort}(|H|))$ I/Os as follows: First we apply procedure CopyVertexLabels and

**Procedure** CONTRACTBIPARTITE (CONTINUED)

16: Create a priority queue $Q$ which is to store directed edges sorted lexicographically.
17: **for** all vertices $v \in V_m$, in increasing order **do**
18: **for** all vertices $w \in V_u$ adjacent to $v$, in increasing order **do**
19: **if** $v$ is the smallest vertex adjacent to $w$ or FINDMIN$(Q) = (v, w)$ **then**
20: **if** FINDMIN$(Q) = (v, w)$ **then**
21: DELETEMIN$(Q)$
22: **end if**
23: **if** ISCONTRACTIBLE$(\{v, w\})$ **then**
24: Contract edge $\{v, w\}$ and make $v$ the parent of $w$ in $F$.
25: **else if** $v$ is not the largest vertex adjacent to $w$ **then**
26: INSERT$(Q, (p_v(w), w))$
27: **end if**
28: **end if**
29: **end for**
30: **end for**
31: Let $G' = (V', E')$ be the resulting graph.

**Algorithm 8.7**
Algorithm 8.6 continued.

scan the edge set of $H$ to extract all edges $\{v, w\}$, $v \in V_m, w \in V_u$. We represent each such edge as a directed edge $(w, v)$. Now we sort these edges lexicographically, thereby storing the edges incident to vertex $w \in V_u$ consecutively, sorted by their endpoints in $V_m$. We scan the resulting list and do the following for every adjacency list $A(w)$, $w \in V_u$: For the first entry $(w, v_1)$, we create a quadruple $(v_1, w, v_2,$ "first"$)$. For all other entries $(w, v_i)$, except the last entry, we create a quadruple $(v_i, w, v_{i+1},$ **nil**$)$. For the last entry $(w, v_k)$, we create a quadruple $(v_k, w,$ **nil**, "last"$)$. We sort the resulting list lexicographically. The result is a concatenation of the sorted adjacency lists of the vertices in $V_m$, where each entry stores the additional information as described in Lines 3-15 of procedure CONTRACTBIPARTITE.

The loop in Lines 17–30 scans this concatenation of adjacency lists and performs a number of priority queue operations. Observe that every edge in $H$ causes at most four priority queue operations to be performed. Since $H$ is planar, there are

only $\mathcal{O}(|H|)$ edges, so that the algorithm performs $\mathcal{O}(|H|)$ priority queue operations, which take $\mathcal{O}(\text{sort}(|H|))$ I/Os using the priority queue from Section 5.1.4.

It remains to show that applying procedure ContractBipartite to graph $H$ leaves no contractible edge incident to a vertex in $V_u$. To see this, observe that the loop in Lines 17–30 inspects all edges incident to vertices in $V_u$, as there are no edges between vertices in $V_u$. For every inspected edge $\{v, w\}$, $v \in V_m$, $w \in V_u$, so that $v$ is the smallest vertex adjacent to $w$ or entry $(v, w)$ is the minimum entry in $Q$, the algorithm contracts edge $\{v, w\}$ unless it is not contractible. Thus, we have to show that $v$ is the smallest vertex adjacent to $w$ or $(v, w)$ is the smallest entry in $Q$, for every edge $\{v, w\}$ whose unmatched endpoint $w$ has not been contracted into another matched vertex at the time when edge $\{v, w\}$ is inspected by the algorithm. We prove this by induction on the number of edge $\{v, w\}$ in the sequence of edges as inspected by the algorithm. For the first edge, vertex $v$ is the matched vertex with smallest number, so that it is also the smallest vertex adjacent to $w$. For every subsequent edge $\{v, w\}$, assume that $v$ is not the smallest vertex adjacent to $w$ because otherwise the claim holds. Then let $v'$ be the predecessor of $v$ in the sorted sequence of vertices adjacent to $w$. Observe that $v' < v$, so that $(v', w) < (v, w)$. Hence, by the induction hypothesis, either $v'$ is the smallest vertex adjacent to $w$ or entry $(v', w)$ is the minimum entry in $Q$ when edge $\{v', w\}$ is visited. As vertex $w$ has not been contracted into a matched vertex when the algorithm inspects edge $\{v, w\}$, edge $\{v', w\}$ is not being contracted at the time when it is inspected by the algorithm. Hence, the algorithm inserts entry $(v, w)$ into $Q$. Now consider entries $(v'', w'') < (v, w)$ queued before the algorithm inspects edge $\{v, w\}$. For each such entry, edge $\{v'', w''\}$ is visited before edge $\{v, w\}$. Moreover, since $(v'', w'') \in Q$, there must be a vertex $v''' < v''$ adjacent to $w''$ which has queued entry $(v'', w'')$. Thus, $v''$ is not the smallest vertex adjacent to $w''$, and edge $\{v'', w''\}$ finds entry $(v'', w'')$ as the minimum entry in $Q$, by the induction hypothesis. Consequently, all entries $(v'', w'') < (v, w)$ are removed from $Q$ before edge $\{v, w\}$ is inspected, so that edge $\{v, w\}$ finds entry $(v, w)$ as the minimum entry in $Q$ when it is inspected. □

As in the proof of Lemma 8.6, let $H^{(1)}, H^{(2)}, \ldots$ be the snapshots of graph $H$ at the beginning of each iteration of the loop in Lines 4–9 of Algorithm 8.4. The following lemma shows that the sizes of these graphs are geometrically decreasing.

**Lemma 8.9** *For all $i \geq 1$, $\left|H^{(i+1)}\right| \leq \frac{1}{2}\left|H^{(i)}\right|$.*

*Proof.* By Lemma 8.8, procedure ContractBipartite guarantees that no contractible edge remains in $H^{(i)}$ which is incident to an unmatched vertex. Hence, $H^{(i+1)}$ contains only vertices that are matched in $H^{(i)}$. In particular, each such vertex is the result of contracting at least one edge in $H^{(i)}$ and hence represents at least two vertices of $H^{(i)}$. □

We summarize this section in the following lemma, which shows that Lines 1–5 of Algorithm 8.1 take $\mathcal{O}(\mathrm{sort}(N))$ I/Os.

**Lemma 8.10** *The sequence of graphs $G_0, \ldots, G_r$ in the graph hierarchy can be constructed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os using linear space.*

*Proof.* By Lemma 8.3, procedure Compress takes $\mathcal{O}(\mathrm{sort}(|G_{i-1}|))$ I/Os to construct graph $G_i$ from graph $G_{i-1}$. By Lemma 8.2(iv), the sizes of graphs $G_0, \ldots, G_r$ are geometrically decreasing. Thus, the whole sequence of graphs $G_0, \ldots, G_r$ can be constructed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. □

## 8.4 The Separator Hierarchy

Having constructed graphs $G_0, \ldots, G_r$, they are used in Lines 6–9 of Algorithm 8.1 to construct a relatively coarse separator of $G$. In particular, this part of the algorithm constructs a separator $S$ of size $\mathcal{O}\left(N/\sqrt{h}\right)$ whose removal partitions $G$ into connected components of size at most $h \log^2(DB)$.

In Line 6, we compute a partition of $G_r$ into subgraphs of size at most $h \log^2(DB)$ as follows: First we use an arbitrary linear-time algorithm to compute a planar embedding of $G_r$, e.g. [31]. Then we apply Theorem 8.1 to compute the desired partition.

Let $S_r = S_r''$ be the computed separator, whose size is $|S_r| \leq c|G_r|/\left(\sqrt{h}\log(DB)\right)$, for some constant $c$ defined in [8].

In the loop in Lines 7–9, we iterate the following procedure until the desired separator $S_0$ for $G_0$ is obtained: Given a separator $S_{j+1}$ for graph $G_{j+1}$, we construct the set $S_j'$ of vertices represented by the vertices in $S_{j+1}$. Then we apply Theorem 8.1 to each connected component of $G_j - S_j'$ whose size exceeds $h\log^2(DB)$. Let $S_j''$ be the set of separator vertices introduced by partitioning the connected components of $G_j - S_j'$ in this manner. Then $S_j = S_j' \cup S_j''$.

**Lemma 8.11** *The separator $S_0$ of $G$ computed in Lines 6–9 of Algorithm 8.1 has size $\mathcal{O}\left(N/\sqrt{h}\right)$. The connected components of $G - S_0$ have size at most $h\log^2(DB)$.*

*Proof.* It follows from the above construction and Lemma 8.2 that

$$\begin{aligned}
|S_0| &\leq \sum_{i=0}^{r} \rho_i |S_i''| \\
&\leq \sum_{i=0}^{r} \rho_i \frac{c|G_i|}{\sqrt{h}\log(DB)} \\
&\leq \sum_{i=0}^{r} \rho_i \frac{28cN}{\rho_i\sqrt{h}\log(DB)} \\
&= \sum_{i=0}^{r} \frac{28cN}{\sqrt{h}\log(DB)} \\
&= \frac{28cN}{\sqrt{h}}.
\end{aligned}$$

The bound on the size of the connected components of $G - S_0$ is explicitly ensured by our construction. □

**Lemma 8.12** *Lines 6–9 of Algorithm 8.1 take $\mathcal{O}(\mathrm{sort}(N))$ I/Os to compute separator $S_0$, provided that $M \geq 56h\log^2(DB)$.*

*Proof.* The computation of separator $S_r$ takes $\mathcal{O}(N/(DB))$ I/Os because graph $G_r$ has size at most $28N/\rho_r = \mathcal{O}(N/(DB))$. Since the sizes of graphs $G_0, \ldots, G_r$ are geometrically decreasing, it is sufficient to show that separator $S_i$ can be constructed

from separator $S_{i+1}$ in $\mathcal{O}(\text{sort}(|G_i|))$ I/Os, in order to prove that the construction of $S_0$ takes $\mathcal{O}(\text{sort}(|G_0|)) = \mathcal{O}(\text{sort}(N))$ I/Os.

Assuming that during the construction of the graph hierarchy, every vertex $v$ in $G_i$ has been labelled with the vertex in $G_{i+1}$ representing $v$, separator $S_i'$ can be constructed in $\mathcal{O}(\text{sort}(|G_i|))$ I/Os: We sort the vertex set $S_{i+1}$, and sort the vertices in $G_i$ by their representatives in $G_{i+1}$. Then we scan the two lists and mark every vertex of $G_i$ as being in $S_i'$ whose representative in $G_{i+1}$ is in $S_{i+1}$. Now it takes $\mathcal{O}(\text{sort}(|G_i|))$ I/Os to compute the connected components of $G_i - S_i'$ using the algorithm of [43].

Since every connected component of $G_{i+1} - S_{i+1}$ has size at most $h\log^2(DB)$, it follows from Lemma 8.2(iii) that every connected component of $G_i - S_i'$ has size at most $56h\log^2(DB) \leq M$. That is, each such connected component fits into internal memory. Thus, we can load each connected component of $G_i - S_i'$ whose size exceeds $h\log^2(DB)$ into internal memory and apply Theorem 8.1 to partition it into subgraphs of size at most $h\log^2(DB)$. As the computation of Theorem 8.1 is carried out in internal memory, partitioning the connected components of $G_i - S_i'$ into subgraphs of size at most $h\log^2(DB)$ takes $\mathcal{O}(\text{scan}(|G_i|))$ I/Os. □

## 8.5 Computing the Final Partition

In order to obtain the final partition, the connected components of $G - S_0$ have to be partitioned into subgraphs of size at most $h$ and boundary size at most $\sqrt{h}$. This is done in Line 10 of Algorithm 8.1. Algorithm 8.8 provides the details of this step.

Algorithm 8.8 first puts all vertices in $S_0$ into separator $S$ and then applies Theorem 8.1 to augment $S$ so that no connected component of $G - S$ has size more than $h$. This is done in Lines 2–7. Lines 19–23 apply Theorem 8.2 to the subgraphs into which the removal of separator $S$ partitions $G$, in order to guarantee that no subgraph of $G - S$ has boundary size exceeding $\sqrt{h}$. However, the preconditions of Theorem 8.2 are not necessarily satisfied by the connected components of $G - S$. In

**Procedure** FINALSEPARATOR

**Input:** A graph $G = (V, E)$ and a separator $S_0$ of size $\mathcal{O}\left(N/\sqrt{h}\right)$ so that no connected component of $G - S_0$ has size exceeding $h \log^2(DB)$.

**Output:** A proper $h$-partition $\mathcal{P} = (S, \mathcal{R})$ of $G$.

1: $S \leftarrow S_0$
2: **for** every connected component $Q$ of $G - S_0$ **do**
3: **if** $|Q| > h$ **then**
4: Apply Theorem 8.1 to compute an $h/|Q|$-separator $S_Q$ of $Q$.
5: $S \leftarrow S \cup S_Q$
6: **end if**
7: **end for**
8: $H \leftarrow (V(H), E(H))$, where

$$V(H) = S \cup \{w_i : Q_i \text{ is a connected component of } G - S\}$$
$$E(H) = \{\{v, w\} \in E : v, w \in S\} \cup \{\{v, w_i\} : v \in S \text{ and } v \in \partial Q_i\}$$

9: **for all** $v \in V(H)$ **do**
10: $\omega(v) \leftarrow \begin{cases} |Q_i| & \text{if } v = w_i \\ 0 & \text{otherwise} \end{cases}$
11: **end for**
12: $H_1 \leftarrow$ MERGELOWDEGREE$(H)$ {ISHEAVY$(v)$ = **true** if $\omega(v) > h/2$ or $v \in S$}
13: $H_2 \leftarrow$ CONTRACTEDGES$(H_1)$ {ISCONTRACTIBLE$(\{v, w\})$ = **true** if $\omega(v) + \omega(w) \leq h$}
14: $H_3 \leftarrow$ MERGELOWDEGREE$(H_2)$ {ISHEAVY$(v)$ = **true** if $\omega(v) > h/2$}
15: **for** every vertex $v \in H_3$ **do**
16: Let $Q'_v = Q_1 \cup \cdots \cup Q_k$, where $w_1, \ldots, w_k$ are the region vertices that have been merged into $v$.
17: **end for**
18: $\mathcal{P} \leftarrow (S, \mathcal{R})$, where $\mathcal{R} = \emptyset$
19: **for** every subgraph $Q'_v$ of $G - S$ **do**
20: $(S_{Q'_v}, \{Q''_1, \ldots, Q''_x\}) \leftarrow$ SEPARATEBOUNDARY$(Q'_v, \partial Q'_v)$
21: $S \leftarrow S \cup S_{Q'_v}$
22: $\mathcal{R} \leftarrow \mathcal{R} \cup \{Q''_1, \ldots, Q''_x\}$
23: **end for**

**Algorithm 8.8**
The details of computing a proper $h$-partition of $G$ in Line 10 of Algorithm 8.1.

particular, there may be as many as $\Omega(N)$ connected components whose total boundary size is $\omega\left(N/\sqrt{h}\right)$, even though the number of separator vertices is $\mathcal{O}\left(N/\sqrt{h}\right)$. The purpose of Lines 8–17 is to group the connected components of $G - S$ into $\mathcal{O}(N/h)$ subgraphs whose total boundary size is $\mathcal{O}\left(N/\sqrt{h}\right)$, so that Theorem 8.2 can be applied.

Lines 8-11 construct a graph $H$ whose vertices represent the connected components of $G - S$ and the vertices in $S$. The edges of $H$ represent their adjacencies. In Lines 12–14, graph $H$ is used to determine which connected components of $G - S$ are to be merged into a single subgraph, in order to satisfy the preconditions of Theorem 8.2. Lines 15–17 merge connected components as determined in Lines 12–14.

In detail, Line 12 merges region vertices of degree at most two in $H$ that are adjacent to the same set of separator vertices. While merging these vertices, the algorithm maintains the invariant that no vertex in $H$ has weight exceeding $h$. We show below that after Line 12, graph $H$ is a planar graph with $\mathcal{O}\left(N/\sqrt{h}\right)$ vertices. Thus, merging the connected components of $G - S$ corresponding to merged region vertices in $H$ produces a partition of $G-S$ into $\mathcal{O}\left(N/\sqrt{h}\right)$ subgraphs. Moreover, the total boundary size of these subgraphs is equal to the number of edges in $H$, which is $\mathcal{O}\left(N/\sqrt{h}\right)$. Thus, all that remains to be done in Lines 13 and 14 is to reduce the number of subgraphs to $\mathcal{O}(N/h)$ by merging subgraphs further. The crucial observation is that merging subgraphs cannot increase the total boundary size, so that after Line 14, we obtain the required partition of $G - S$ into $\mathcal{O}(N/h)$ subgraphs with total boundary size $\mathcal{O}\left(N/\sqrt{h}\right)$. As Lines 12–14 only merge region vertices in $H$, the corresponding connected components of $G - S$ need to be merged, in order to obtain the desired subgraphs. This is done in Lines 15–17.

Even though the partition of $G-S$ into subgraphs now satisfies the preconditions of Theorem 8.2, this theorem cannot be applied directly because the algorithm of [73] needs to work with the subgraph $\bar{Q} = G[V(Q) \cup \partial Q]$ of $G$, for every subgraph $Q$ of $G - S$ to be partitioned. Even though $|Q| \leq h$, the size of $\bar{Q}$ may be as large as $\Omega\left(N/\sqrt{h}\right)$ because the construction so far does not give a better bound on the

**Procedure** SEPARATEBOUNDARY

**Input:** A subgraph $\bar{Q} = G[V(Q) \cup \partial Q]$ of $G$.
**Output:** A proper $h$-partition $\mathcal{P}_Q = (S_Q, \mathcal{R})$ of $Q$.

1: Assign weight 1 to every vertex in $\partial Q$ and weight 0 to every vertex in $V(Q)$.
2: Apply procedure MERGELOWDEGREE to the bipartite subgraph $H$ of $\bar{Q}$ induced by the edges in $\bar{Q}$ having one endpoint in $V(Q)$ and the other in $\partial Q$. This merges all vertices of degree at most two in $\partial Q$ that are adjacent to the same vertices in $V(Q)$ into a single vertex. In the terminology of Algorithm 8.3, this means that a vertex is *light* if it is in $\partial Q$ and of degree at most two. All other vertices are *heavy*.
3: Apply Theorem 8.2 to the resulting graph $\tilde{Q}$ to compute a 1/2-proper $h$-partition $P_Q = (S_Q, \{Q_1, \ldots, Q_x\})$ of $\tilde{Q}$.
4: For every separator vertex $v \in S_Q$ of weight more than one, add its adjacent vertices in $V(Q)$ to $S_Q$ and remove these vertices from the graph $Q_i$ containing these vertices.

**Algorithm 8.9**
Partitioning subgraphs of size at most $h$ into subgraphs of boundary size at most $\sqrt{h}$.

boundary size of each particular subgraph. Thus, graph $\bar{Q}$ may not fit into internal memory. Algorithm 8.9 shows how to apply Theorem 8.2 to graph $\bar{Q}$ in an I/O-efficient manner. In order to do that, it replaces separator vertices of degree at most two which are adjacent to the same vertices in $Q$ with a single separator vertex. The weight of this vertex is the same as the number of separator vertices it represents. This compresses $\partial Q$ so that $|\partial Q| \leq 6|Q|$. Hence, the resulting graph $\tilde{Q}$ fits into internal memory, and Theorem 8.2 can be applied to $\tilde{Q}$ in internal memory.

Even though every subgraph $K$ of $\tilde{Q}$ in the partition produced by applying Theorem 8.2 to $\tilde{Q}$ has boundary size at most $\sqrt{h}/2$, the corresponding subgraph of $\bar{Q}$ may have a large boundary. This happens if at least one of the vertices on the boundary of $K$ has large weight. However, every vertex of large weight in $\tilde{Q}$ is adjacent to at most two vertices in $Q$. Thus, each such vertex in the separator can be replaced with its two adjacent vertices in $Q$. This increases the total size of the separator by a factor of at most two and guarantees that no subgraph in the partition of $Q$ corresponding to the computed partition of $\tilde{Q}$ has boundary size exceeding $\sqrt{h}$.

The following two lemmas show that Algorithm 8.8 produces the desired partition, and that it does so in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.

**Lemma 8.13** *Given a planar graph $G = (V, E)$ and a separator $S_0 \subseteq V$ of size $\mathcal{O}\left(N/\sqrt{h}\right)$ whose removal partitions $G$ into connected components of size at most $h \log^2(DB)$, Algorithm 8.8 computes a proper $h$-partition $\mathcal{P} = (S, \mathcal{R})$ of $G$.*

*Proof.* Lines 2–7 of Algorithm 8.8 apply Theorem 8.1 to every connected component $Q$ of $G - S$ whose size exceeds $h$, in order to partition it into subgraphs of size at most $h$. The vertices in the computed separator $S_Q$ of $Q$ are added to $S$. By Theorem 8.1, this introduces $\mathcal{O}\left(|Q|/\sqrt{h}\right)$ separator vertices per connected component $Q$, $\mathcal{O}\left(N/\sqrt{h}\right)$ in total. Hence, the size of separator $S$ after Line 7 is $\mathcal{O}\left(N/\sqrt{h}\right)$.

To make the algorithm I/O-efficient, the merging of connected components of $G - S$ into subgraphs $Q_1, \ldots, Q_k$ is split into two parts: Lines 12–14 determine which connected components are to be merged into the same subgraph. Lines 15–17 then merge these subgraphs. To simplify the argument, we assume that when two region vertices in $H$ are merged, the corresponding connected components of $G - S$ are merged immediately, instead of postponing the merging of connected components to Lines 15–17.

Under this assumption, Line 12 merges connected components of weight no more than $h/2$ and boundary size at most two which are adjacent to the same set of separator vertices. That is, after Line 12, $G - S$ is partitioned into $\mathcal{O}\left(N/\sqrt{h}\right)$ subgraphs: $\mathcal{O}\left(N/h\right)$ subgraphs of size at least $h/2$ and $\mathcal{O}\left(N/\sqrt{h}\right)$ subgraphs of size less than $h/2$, by Corollary 8.1.

Graph $H$ is planar and has $\mathcal{O}\left(N/\sqrt{h}\right)$ vertices after the computation in Line 12: $\mathcal{O}\left(N/\sqrt{h}\right)$ separator vertices and $\mathcal{O}\left(N/\sqrt{h}\right)$ region vertices. A region vertex $w$ in $H$ is adjacent to a separator vertex $v$ in $H$ if and only if $v \in \partial Q$, where $Q$ is the subgraph of $G - S$ represented by $w$. That is, the total boundary size of the subgraphs of $G - S$ obtained by merging the connected components in Line 12 is $\mathcal{O}\left(N/\sqrt{h}\right)$, the same as the number of edges in $H$.

Lines 13–14 now apply procedures ContractEdges and MergeLowDegree to guarantee that no two light vertices in $H$ are adjacent to each other, and no two light vertices of degree at most two are adjacent to the same set of heavy vertices, where a vertex is *light* if its weight is at most $h/2$. It follows from the same arguments as applied in the proof of Lemma 8.2 that the resulting compressed version of $H$ has size $\mathcal{O}(N/h)$. In particular, the corresponding partition of $G - S$ consists of $\mathcal{O}(N/h)$ subgraphs of size at most $h$. Merging subgraphs in this manner cannot increase the total boundary size, so that this produces a partition of $G-S$ into $\mathcal{O}(N/h)$ subgraphs of size at most $h$ and total boundary size $\mathcal{O}\left(N/\sqrt{h}\right)$. Thus, the partition obtained after Line 17 of Algorithm 8.8 satisfies the conditions of Theorem 8.2.

As procedure SeparateBoundary produces a proper $h$-partition of the graph $Q$ to which it is applied, the final partition $\mathcal{P} = (S, \{G_1, \ldots, G_k\})$ computed by Algorithm 8.8 has the property that $\partial G_i \leq \sqrt{h}$, for $1 \leq i \leq k$. In order to show that $\mathcal{P}$ is proper, it thus suffices to show that $k = \mathcal{O}(N/h)$. However, as the number of subgraphs before Line 19 is $\mathcal{O}(N/h)$, the total number of subgraphs obtained at the end of the algorithm is $\mathcal{O}(N/h)$, by Theorem 8.2. □

**Lemma 8.14** *Algorithm 8.8 takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os and uses linear space, provided that $M \geq h \log^2(DB)$.*

*Proof.* The implementation of Lines 2–7 of Algorithm 8.8 takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os: Computing the connected components of $G - S$ takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os using the algorithm of [43]. Once the connected components are identified, the partitioning of each component can be carried out in internal memory, so that partitioning all components takes $\mathcal{O}(\mathrm{scan}(N))$ I/Os.

Given that every vertex in $G - S$ is labelled with a label identifying its connected component, graph $H$ can be computed by replacing every vertex and every edge endpoint in $G - S$ with its component label and then removing duplicates and loops from the resulting vertex and edge sets. This takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os. Lines 12–14 take $\mathcal{O}(\mathrm{sort}(N))$ I/Os, by Lemmas 8.4 and 8.6.

In order for Lines 15–17 to be able to carry out their computation, Lines 8–14 have to set up the appropriate information. In particular, Line 8 labels every vertex $v$ in a connected component $Q$ of $G-S$ with the name $\nu(v)$ of the region vertex $w$ in $H$ representing $Q$. Procedures MERGELOWDEGREE and CONTRACTEDGES construct a forest $F$ representing the contraction history of the vertices in $H$. In Lines 15–17, we compute the root $r(v)$ of the tree in $F$ containing vertex $v$, for every vertex $v \in F$, using the following strategy: First we find the connected components of $F$ and use the Euler tour technique and list-ranking to label all vertices in every subtree with the name of the root of the tree. Then we sort the vertices in $G-S$ by their labels $\nu(v)$ and the vertices in $F$ by their names. Now a single scan of these two sorted lists is sufficient to replace every label $\nu(v)$, $v \in G-S$, with the label $r(\nu(v))$. This assigns the same label to all vertices in the same subgraph of $G-S$. Given this labelling, the vertices and edges in $G-S$ can be sorted by their labels so that the vertices and edges of each graph $Q$ are stored consecutively. For each such subgraph $Q$, we find the edges connecting vertices in $Q$ to vertices in $S$. Adding these edges to $E(Q)$ and their endpoints to $V(Q)$ produces graph $\bar{Q}$.

Computing the connected components of $F$ takes $\mathcal{O}(\text{sort}(N))$ I/Os using the algorithm of [43]. Labelling every vertex in $F$ with the root of its tree in $F$ takes $\mathcal{O}(\text{sort}(N))$ I/Os, as list-ranking takes $\mathcal{O}(\text{sort}(N))$ I/Os [43]. The remainder of the implementation of Lines 15–17 requires the sorting and scanning of sets of size $\mathcal{O}(N)$, so that the rest of the construction in Lines 15–17 takes $\mathcal{O}(\text{sort}(N))$ I/Os as well.

The application of Algorithm 8.9 to a single graph $\bar{Q}$ in Line 19 of Algorithm 8.8 takes $\mathcal{O}(\text{sort}(|\bar{Q}|))$ I/Os: Line 1 can be implemented in a single scan of the vertex set of $\bar{Q}$. Line 2 takes $\mathcal{O}(\text{sort}(|\bar{Q}|))$ I/Os, by Lemma 8.4. Line 3 is carried out in internal memory. Adding the vertices in $V(Q)$ adjacent to vertices in $S_Q$ of weight more than one to $S_Q$ and removing them from the graphs $Q_1, \ldots, Q_x$ are the same operation, as containment of these vertices in $S_Q$ or graphs $Q_1, \ldots, Q_x$ is represented by labelling every vertex appropriately. Now, in order to add these vertices to $S_Q$, we apply operation COPYVERTEXLABELS to label all edges that have an endpoint in $S_Q$

whose weight is more than one. Then we scan the edge list to extract all these edges and apply operation SumEdgeLabels to add all endpoints of these edges to $S_Q$.

As the total boundary size of all subgraphs $Q$ is $\mathcal{O}\left(N/\sqrt{h}\right)$, the total size of all graphs $\bar{Q}$ is $\mathcal{O}(N)$. Hence, Lines 19–23 of Algorithm 8.8 take $\mathcal{O}(\mathrm{sort}(N))$ I/Os. □

We are now ready to state the main result of this chapter.

**Theorem 8.3** *Given a planar graph $G$ and an integer $h > 0$, Algorithm 8.1 computes a proper $h$-partition of $G$. The algorithm takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os and linear space, provided that $M \geq 56h\log^2(DB)$.*

*Proof.* The I/O-complexity of Algorithm 8.1 follows from Lemmas 8.10, 8.12, and 8.14. The correctness of the algorithm follows from Lemmas 8.11 and 8.13. □

# Chapter 9

# Planarity Testing and Planar Embedding

Even though the separator algorithm presented in Chapter 8 does not require a planar embedding, many algorithms for planar graphs, including the applications presented in Chapters 10 and 11, exploit the information provided by such an embedding. In this chapter, we provide an I/O-efficient algorithm which tests whether a given graph is planar and if so, computes a planar embedding of the graph. The algorithm exploits the fact that our separator algorithm does not require a planar embedding to be given as part of the input. Hence, we can use the separator algorithm to compute a normal $(DB)^2$-partition $\mathcal{P} = (S, \{G_1, \ldots, G_k\})$ of $G$. We compute a planar embedding of $G$ from planar embeddings of graphs $G_1, \ldots, G_k$ and a planar embedding of a graph $A$ which is obtained from $G$ by replacing graphs $G_1, \ldots, G_k$ with "constraint graphs" $C_1, \ldots, C_k$ whose total size is $\mathcal{O}(N/(DB))$.

In Section 9.1, we prove a convenient characterization of triconnected planar graphs, which will be handy whenever we have to show that a constraint graph we construct is triconnected and hence has a unique planar embedding. In Sections 9.2 through 9.8, we present our algorithm for planarity testing and planar embedding. By Lemma 2.1, the I/O-complexity of our embedding algorithm can be reduced to $\mathcal{O}(\text{perm}(N))$. In Section 9.9, we prove a matching lower bound.

## 9.1 Triconnected Planar Graphs: A Characterization

If a planar graph $G$ is triconnected, the task of capturing all possible embeddings of $G$ is simplified considerably by the following classical result, which states that in this case, the planar embedding of $G$ is unique. We exploit this fact in Section 9.4, where our construction captures the characteristics of the embeddings of the tricomps of a planar graph.

**Theorem 9.1 (Whitney [174])** *Let $\hat{G}_1$ and $\hat{G}_2$ be two planar embeddings of a triconnected simple planar graph $G$. Let $f_1, \ldots, f_k$ be the faces of $\hat{G}_1$, and $f'_1, \ldots, f'_l$ be the faces of $\hat{G}_2$. Then $k = l$, and there exists a permutation $\sigma : [1, k] \to [1, k]$ so that faces $f_i$ and $f'_{\sigma(i)}$ have the same vertices on their boundaries, for all $1 \leq i \leq k$. Moreover, either the boundary vertices of faces $f_i$ and $f'_{\sigma(i)}$ appear in the same order around both faces, for all $1 \leq i \leq k$, or their orders are reversed, for all $1 \leq i \leq k$.*

Given a planar embedding $\hat{G}$ of a planar graph $G$, we use the following lemma to prove that graph $G$ is triconnected.

**Lemma 9.1** *Let $G$ be a simple biconnected planar graph which is not a cycle, and let $\hat{G}$ be a planar embedding of $G$. Graph $G$ is triconnected if and only if there are no two faces $f_1$ and $f_2$ of $\hat{G}$ and two vertices $v$ and $w$ such that $v$ and $w$ are on the boundary of both $f_1$ and $f_2$, but edge $\{v, w\}$ does not exist or is on the boundary of at most one of $f_1$ and $f_2$.*

*Proof.* First assume that there are two faces $f_1$ and $f_2$ of $\hat{G}$ and two vertices $v$ and $w$ such that $v$ and $w$ are on the boundaries of both $f_1$ and $f_2$, but edge $\{v, w\}$ does not exist or is on the boundary of at most one of $f_1$ and $f_2$. Assume w.l.o.g. that edge $\{v, w\}$ is on the boundary of face $f_2$ if this edge exists (Figure 9.1a). Let $P_1$ be the path from $v$ to $w$ counterclockwise around $f_1$, and $P_2$ be the path from $v$ to $w$ clockwise around $f_1$. As edge $\{v, w\}$ is not on the boundary of face $f_1$, path $P_1$ contains an internal vertex $x$, and path $P_2$ contains an internal vertex $y$. We can connect vertices $v$ and $w$ by two curves $\alpha$ and $\beta$ that are completely contained in

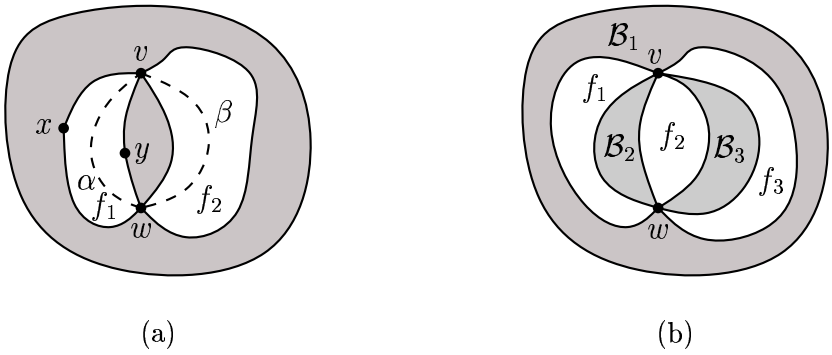



**Figure 9.1**
Proof of Lemma 9.1.

the interiors of faces $f_1$ and $f_2$, respectively. The union of curves $\alpha$ and $\beta$ is a closed Jordan curve $\gamma$ which does not intersect any edges of $G$ and contains only vertices $v$ and $w$. Vertex $x$ is outside $\gamma$, vertex $y$ is inside. Thus, any path from $x$ to $y$ must contain either $v$ or $w$, so that $\{v, w\}$ is a separation pair. Hence, graph $G$ cannot be triconnected.

Now assume that $G$ is not triconnected. Then $G$ must contain a separation pair $\{v, w\}$. Let $B_1, \ldots, B_k$ be the bridges of $\{v, w\}$. At least two of the bridges are non-trivial. Let $\hat{B}_1, \ldots, \hat{B}_k$ be the planar embeddings of $B_1, \ldots, B_k$ induced by $\hat{G}$. Then every embedding $\hat{B}_i$ has exactly one face containing all other embeddings $\hat{B}_1, \ldots, \hat{B}_{i-1}, \hat{B}_{i+1}, \ldots, \hat{B}_k$. Hence, embedding $\hat{G}$ looks like in Figure 9.1b. If $\{v, w\} \notin G$, faces $f_1$ and $f_2$ do not share edge $\{v, w\}$, but they share vertices $v$ and $w$. If $\{v, w\} \in G$, $k \geq 3$ and w.l.o.g. $B_k = (\{v, w\}, \{\{v, w\}\})$. Then again, faces $f_1$ and $f_2$ do not share edge $\{v, w\}$. □

## 9.2 Overview of the Algorithm

In this section, we give an overview of our I/O-efficient algorithm for planarity testing and planar embedding. The algorithm exploits the fact that the separator algorithm of the previous section does not require a planar embedding of the graph. In fact,

graph $G$ does not even have to be planar as long as the following three conditions are satisfied:

(i) Graph $G_r$ in the graph hierarchy is planar,

(ii) $|G_i| \leq 28cN/\rho_i$, for all $1 \leq i \leq r$ and the same constants $c$ and $\rho_i$ as defined in Chapter 8, and

(iii) The connected components of $G_i - S_i'$ are planar, for all $0 \leq i < r$.

Given that these conditions are satisfied, the separator algorithm produces a small separator in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. Also, the algorithm can easily be augmented to test for violation of one of these conditions. If it finds that one of the conditions is violated, the algorithm can abort and report that $G$ is non-planar, as we have shown in the Chapter 8 that a planar graph satisfies these conditions.

Now the idea of our algorithm is quite simple: We compute a proper $(DB)^2$-partition $\mathcal{P} = (S, \{G_1, \ldots, G_k\})$ of $G$ and test whether graphs $G_1, \ldots, G_k$ are planar. If so, we replace each graph $G_i$ with a "constraint graph" $C_i$ of size $\mathcal{O}(DB)$ so that the resulting "approximate graph" $A$ is planar if and only if $G$ is planar. The planarity of $A$ can be tested in $\mathcal{O}(N/(DB))$ I/Os, as $A$ has size at most $\mathcal{O}(N/(DB))$. A planar embedding of $G$ can be obtained from a planar embedding of $A$ by replacing the induced embeddings of graphs $C_1, \ldots, C_k$ with "consistent" embeddings of graphs $G_1, \ldots, G_k$.

Algorithm 9.1 provides an outline of our algorithm. Sections 9.3 to 9.7 describe the construction of constraint graphs $C_1, \ldots, C_k$ from graphs $G_1, \ldots, G_k$ and show that the approximate graph $A$ has the desired properties. Section 9.8 shows how to derive a "consistent" planar embedding $\hat{G}_i$ of $G_i$ from a planar embedding $\hat{C}_i$ of $C_i$ and how to combine these embeddings to obtain a planar embedding of $G$.

**Theorem 9.2** *Algorithm 9.1 correctly tests whether a simple graph $G$ is planar and if so, computes a planar embedding $\hat{G}$ of $G$. The algorithm takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, provided that $M \geq (DB)^2 \log^2(DB)$, and uses linear space.*

**Procedure** PLANAREMBEDDING

**Input:** A simple graph $G = (V, E)$.
**Output:** A planar embedding $\hat{G}$ of $G$ or the answer that $G$ is not planar.

1: **if** $|E| > 3|V| - 6$ **then**
2: Report that $G$ is not planar and exit.
3: **end if**
4: Compute a proper $(DB)^2$-partition $\mathcal{P} = (S, \{G_1, \ldots, G_k\})$ of $G$.
5: **if** Line 4 reports an error **then**
6: Report that $G$ is not planar and exit.
7: **end if**
8: Let $\tilde{G}_1, \ldots, \tilde{G}_k$ be the graphs defined as

$$\tilde{G}_i = (V(G_i) \cup \partial G_i, \{\{v, w\} \in E : v \in V(G_i) \wedge w \in V(G_i) \cup \partial G_i\}).$$

9: Let $G'_1, \ldots, G'_l$ be the connected components of graphs $\tilde{G}_1, \ldots, \tilde{G}_k$.
10: **for** $j = 1, \ldots, l$ **do**
11: **if** $G'_j$ is not planar **then**
12: Report that $G$ is not planar and exit.
13: **end if**
14: Compute the constraint graph $C_j$ of $G'_j$.
15: **end for**
16: Let $A = G[S] \cup C_1 \cup \cdots \cup C_l$
17: **if** $A$ is not planar **then**
18: Report that $G$ is not planar and exit.
19: **end if**
20: Compute a planar embedding $\hat{A}$ of $A$.
21: **for** $i = 1, \ldots, j$ **do**
22: Let $\hat{C}_j$ be the restriction of embedding $\hat{A}$ to $C_j$.
23: Replace $\hat{C}_j$ by a consistent embedding $\hat{G}'_j$ of $G'_j$.
24: **end for**
25: Let $\hat{G}$ be the resulting embedding of $G$.

**Algorithm 9.1**
An algorithm that tests whether a given graph $G = (V, E)$ is planar and if so, computes a planar embedding of $G$.

*Proof.* First we prove the correctness of Algorithm 9.1. Assume that our algorithm reports that graph $G$ is not planar. This can happen in Lines 2, 6, 12, or 18. If $|E| > 3|V| - 6$, graph $G$ is not planar, by Euler's formula. As argued at the beginning of this section, $G$ cannot be planar if Algorithm 8.1 fails to produce a small separator of $G$. If one of the graphs $G'_1, \ldots, G'_l$ is non-planar, $G$ cannot be planar, as these are subgraphs of $G$. Finally, if $A$ is non-planar, $G$ cannot be planar, by Lemma 9.15. On the other hand, if our algorithm does not report that $G$ is non-planar, the output of the algorithm is a planar embedding $\hat{G}$ of $G$, by Lemma 9.17.

Next we analyze the I/O-complexity of Algorithm 9.1. Lines 1–3 of the algorithm take $\mathcal{O}(\text{scan}(N))$ I/Os, as they only require counting the vertices and edges in $G$. Given that $M \geq (DB)^2 \log^2(DB)$, Theorem 8.3 ensures that Lines 4–7 take $\mathcal{O}(\text{sort}(N))$ I/Os. The computation of graphs $\tilde{G}_1, \ldots, \tilde{G}_k$ in Line 8 of the algorithm takes $\mathcal{O}(\text{sort}(N))$ I/Os: First we use procedure CopyVertexLabels to label every edge with the names of the subgraphs among $G[S], G_1, \ldots, G_k$ containing its endpoints. Then we sort the edge list lexicographically by these labels. This stores the edges of each graph $\tilde{G}_i$ consecutively if we assume that edges with only one endpoint in $G[S]$ are sorted primarily by their other endpoints. Now we scan this list to produce for each set $E(\tilde{G}_i)$, a list of the endpoints of the edges in $E(\tilde{G}_i)$. By applying operation DuplicateRemoval to each of these vertex lists, we obtain the vertex sets $V(\tilde{G}_1), \ldots, V(\tilde{G}_k)$ of graphs $\tilde{G}_1, \ldots, \tilde{G}_k$. Line 9 takes $\mathcal{O}(\text{scan}(N))$ I/Os, as each subgraph $\tilde{G}_i$ has size at most $(DB)^2 + (DB)$. Thus, each subgraph fits into internal memory, and graphs $G'_1, \ldots, G'_l$ can be computed by loading graphs $\tilde{G}_1, \ldots, \tilde{G}_k$ into internal memory, one at a time, and partitioning each of them into its connected components. Similarly, Lines 10–15 take $\mathcal{O}(\text{scan}(N))$ I/Os, as each subgraph $G'_j$ is small enough to fit into internal memory. Line 16 requires an application of procedure DuplicateRemoval to eliminate multiple vertices and edges with the same name in different subgraphs. Thus, this step takes $\mathcal{O}(\text{sort}(N))$ I/Os. By Lemma 9.16, graph $A$ has size $\mathcal{O}(N/(DB))$, so that Lines 17–20 of the algorithm

take $\mathcal{O}(N/(DB))$ I/Os using for instance the linear-time planarity algorithm of [31]. Lines 21–25 take $\mathcal{O}(\text{sort}(N))$ I/Os, by Lemma 9.17. □

## 9.3 Computing the Constraint Graphs

The core of our algorithm is the construction of the constraint graphs $C_1, \ldots, C_l$ from graphs $G'_1, \ldots, G'_l$. The construction ensures that graph $G[G'_j/C_j]$ is planar if and only if $G$ is planar. A planar embedding $\hat{G}$ of $G$ can be obtained from a planar embedding $\hat{G}[G'_j/C_j]$ of $G[G'_j/C_j]$ by locally replacing the embedding of $C_j$ induced by $\hat{G}[G'_j/C_j]$ with a consistent embedding of $G'_j$.

We assume henceforth that graphs $G'_1, \ldots, G'_l$ are planar because otherwise Algorithm 9.1 correctly reports that $G$ is non-planar, regardless of the correctness of the rest of the algorithm. The construction of graphs $C_1, \ldots, C_l$ partitions each graph $G'_j$ into its bicomps $\mathcal{B}_{j,1}, \ldots, \mathcal{B}_{j,q_j}$, and each bicomp $\mathcal{B}_{j,k}$ into its tricomps $\mathcal{T}_{j,k,1}, \ldots, \mathcal{T}_{j,k,r_{j,k}}$, using linear-time algorithms of [97, 163]. The constraint graph $C_j$ of $G'_j$ is now constructed in a bottom-up fashion from the constraint graphs $C_{\mathcal{T}_{j,k,l}}$ of tricomps $\mathcal{T}_{j,k,l}$. In particular, the constraint graph $C_{\mathcal{B}_{j,k}}$ of a bicomp $\mathcal{B}_{j,k}$ is computed by classifying the tricomps of $\mathcal{B}_{j,k}$ into two classes: "Essential" tricomps are replaced by their constraint graphs. "Inessential" tricomps are either completely removed, or groups of them are replaced by constraint graphs of constant size. The construction of $C_j$ from constraint graphs $C_{\mathcal{B}_{j,1}}, \ldots, C_{\mathcal{B}_{j,q_j}}$ follows the same pattern: "Essential" bicomps are replaced by their constraint graphs. "Inessential" bicomps are either removed, or groups of them are replaced by constraint graphs of constant size.

The classification of subgraphs as essential or inessential is closely tied to the concept of required vertices in such a subgraph. For any subgraph $H$ of $G$, the *required* vertices of $H$ are the vertices shared by $H$ and $G-H$. That is, for a graph $G'_j$, all separator vertices in $G'_j$ are required. For a bicomp $\mathcal{B}$, all separator vertices and cutpoints in $\mathcal{B}$ are required. Finally, for a tricomp $\mathcal{T}$, all separator vertices, cutpoints, and members of separation pairs are required. For any graph $H$, the vertex set of its constraint graph $C_H$ has to contain at least the required vertices of $H$. To see why

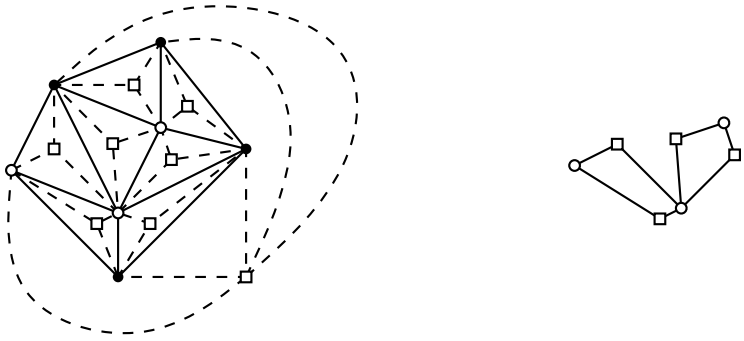
this is necessary, assume that there exists a path in $H$ connecting two of its required vertices. If this path is part of a subgraph of $G$ homeomorphic to $K_5$ or $K_{3,3}$, and one of the two required vertices is not present in $C_H$, $G[H/C_H]$ may be planar even though $G$ is not.

The following sections follow the bottom-up construction of graphs $C_1, \ldots, C_l$. Section 9.4 describes the construction of the constraint graph $C_\mathcal{T}$ of a tricomp $\mathcal{T}$. Section 9.5 shows how to construct the constraint graph $C_\mathcal{B}$ of a bicomp $\mathcal{B}$, using the constraint graphs of the tricomps of $\mathcal{B}$ as building blocks. Section 9.6 describes how to assemble the constraint graph $C_j$ of graph $G'_j$ from the constraint graphs of its bicomps.

## 9.4 The Constraint Graph of a Tricomp

Let $\mathcal{T}$ be a tricomp, and let $R(\mathcal{T})$ denote its set of required vertices. For any graph $H$ containing virtual edges, we define the *kernel* $H^\circ$ of $H$ as the graph obtained from $H$ by removing these edges. Our goal is to construct a constraint graph $C_\mathcal{T}$ for $\mathcal{T}$ whose vertex set has size $\mathcal{O}(|R(\mathcal{T})|)$, which contains all virtual edges of $\mathcal{T}$, and such that $G$ is planar if and only if $G[\mathcal{T}^\circ/C_\mathcal{T}^\circ]$ is planar. If $\mathcal{T}$ is a bond, the constraint graph $C_\mathcal{T}$ of $\mathcal{T}$ is $\mathcal{T}$ itself. If $\mathcal{T}$ is a cycle $\mathcal{T} = (v_1, \ldots, v_k)$, let $v'_1, \ldots, v'_l$ be the required vertices in $\mathcal{T}$, appearing in this order along $\mathcal{T}$. Then $C_\mathcal{T}$ is the cycle $C_\mathcal{T} = (v'_1, \ldots, v'_l)$. The rest of this section deals with the construction of $C_\mathcal{T}$ in the case when $\mathcal{T}$ is a triconnected simple graph.

Since $\mathcal{T}$ is a triconnected simple graph, the planar embedding $\hat{\mathcal{T}}$ of $\mathcal{T}$ is unique. The *face-on-vertex graph* $G_F$ of $\hat{\mathcal{T}}$ is defined as follows (Figure 9.2a): $G_F$ contains all vertices of $\mathcal{T}$ as well as one vertex $f^*$ per face $f$ in $\hat{\mathcal{T}}$. There is an edge between a face vertex $f^*$ and a vertex $w \in \mathcal{T}$ if and only if $w$ appears on the boundary of face $f$. Graph $G_F$ is planar. A planar embedding $\hat{G}_F$ of $G_F$ is obtained by ordering edges $\{f^*, w_1\}, \ldots, \{f^*, w_k\}$ around $f^*$ in the same order as vertices $w_1 \ldots, w_k$ along the boundary of face $f$ in $\hat{\mathcal{T}}$.



(a) (b)

**Figure 9.2**
(a) A triconnected simple graph $\mathcal{T}$ with its face-on-vertex graph $G_F$. Required vertices are white discs. Vertices that are not required are black discs. Face vertices are squares. Edges of $\mathcal{T}$ are solid. Edges of $G_F$ are dashed. (b) The reduced face-on-vertex graph $G'_F$ of $\mathcal{T}$.

Our goal is to construct $C_{\mathcal{T}}$ so that the order of required vertices around the faces of $\hat{\mathcal{T}}$ is preserved in any embedding of $C_{\mathcal{T}}$. First we remove all vertices in $V(\mathcal{T}) \setminus R(\mathcal{T})$ from $G_F$. Then we remove face vertices adjacent to at most one required vertex from $G_F$, but ensure that the degree of any required vertex in $G_F$ remains at least two. We call the resulting graph $G'_F$ the *reduced face-on-vertex graph* of $\hat{\mathcal{T}}$ (see Figure 9.2b).

**Lemma 9.2** *The reduced face-on-vertex graph $G'_F$ of a tricomp $\mathcal{T}$ is planar and has at most $10|R(\mathcal{T})|$ vertices.*

*Proof.* As $G'_F$ is a subgraph of $G_F$, the planarity of $G'_F$ is obvious. In fact, we use the embedding $\hat{G}'_F$ of $G'_F$ induced by the embedding $\hat{G}_F$ of $G_F$ in the remainder of this section. In order to count the number of vertices in $G'_F$, we partition the vertices in $R(\mathcal{T})$ into two groups. The vertices in the first group, $R_1$, are adjacent only to face vertices of degree one. The vertices in the second group, $R_2$, have at least one neighbor of degree at least two.

The total number of face vertices adjacent to vertices in $R_1$ is $2|R_1|$. Every vertex in $R_2$ has at most one adjacent face vertex of degree one. In order to count the

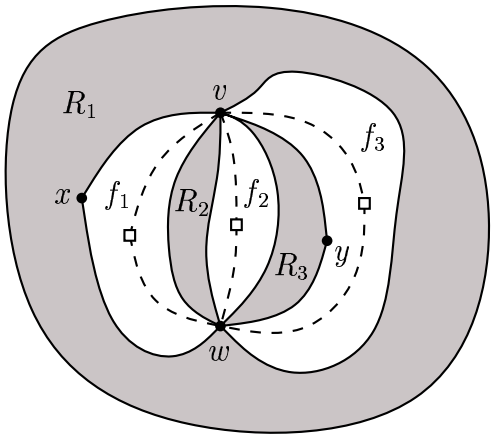
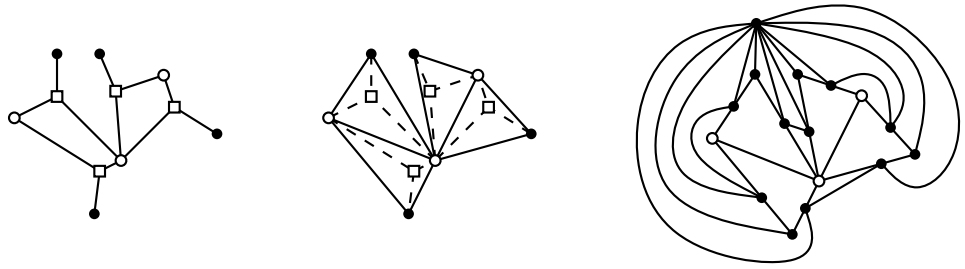



**Figure 9.3**
Proof of Lemma 9.2.

face vertices of degree two in $G'_F$, we consider the subgraph $H$ of $G'_F$ induced by all vertices in $R_2$ and all face vertices of degree two in $G'_F$. We construct a graph $H'$ containing all vertices in $R_2$. There is an edge $\{v, w\} \in H'$ if there exists a face vertex in $H$ which is adjacent to $v$ and $w$. As $H$ is a subgraph of $G'_F$, $H$ is planar. A planar embedding of $H'$ can easily be derived from a planar embedding of $H$. Hence, $H'$ has at most $3|R_2|$ edges. We associate a face vertex $x$ of degree two with edge $\{v, w\} \in H'$ if $x$ is adjacent to $v$ and $w$ in $H$. We show that there are at most two face vertices associated with each edge in $H'$, so that there are at most $6|R_2|$ face vertices of degree two in $G'_F$.

Assume that there are two vertices $v$ and $w$ so that edge $\{v, w\}$ in $H'$ has three face vertices $f_1^*$, $f_2^*$, and $f_3^*$ associated with it (see Figure 9.3). Then $\mathbb{R}^2 \setminus (v \cup w \cup f_1 \cup f_2 \cup f_3)$ consists of three disjoint regions $R_1$, $R_2$, and $R_3$. Only one of these regions can be degenerate, i.e., consist only of the embedding of edge $\{v, w\}$. W.l.o.g., assume that $R_2$ is this region. Then $R_1$ contains some vertex $x$, and $R_3$ contains some vertex $y$. Any path from $x$ to $y$ must pass through either $v$ or $w$, as there are no edges crossing faces $f_1$, $f_2$, and $f_3$. Thus, $\{v, w\}$ is a separation pair, contradicting the triconnectivity of $\mathcal{T}$.



(a) (b) (c)

**Figure 9.4**
(a) The reduced face-on-vertex graph $G'_F$ of tricomp $\mathcal{T}$ shown in Figure 9.2a, augmented with dummy vertices. (b) The graph $H$ whose face-on-vertex graph has the augmented face-on-vertex graph $G'_F$ as a subgraph. (c) The constraint graph $C_{\mathcal{T}}$ of $\mathcal{T}$.

The subgraph $H''$ of $G'_F$ induced by all vertices in $R_2$ and all face vertices of degree at least three is planar and bipartite, where $V_1 = R_2$ and $V_2$ contains all face vertices of degree at least three. Hence, by Lemma 8.1, $|V_2| \leq 2|V_1| = 2|R_2|$. Thus, $G'_F$ has at most $(|R_1| + 2|R_1|) + (|R_2| + |R_2| + 6|R_2| + 2|R_2|) \leq 10|R(\mathcal{T})|$ vertices. □

Next we use graph $G'_F$ to construct the constraint graph $C_{\mathcal{T}}$ of $\mathcal{T}$. In order to construct $C_{\mathcal{T}}$, we augment $G'_F$ so that every face vertex in $G'_F$ has degree at least three. For each face vertex $f^*$ in $G'_F$ of degree at most two, we add one or two dummy vertices to $G'_F$ and make them adjacent to $f^*$. If $f^*$ has degree two, let $x$ and $y$ be the two required vertices adjacent to $f^*$ in $G'_F$. If there is an edge between $x$ and $y$ in $\mathcal{T}$, assume that $x$, edge $\{x, y\}$, and $y$ appear in this order clockwise along the boundary of face $f$. Then we add dummy vertex $z$ incident to $f^*$ between $y$ and $x$ in the clockwise order of vertices around vertex $f^*$. This construction is illustrated in Figure 9.4a for the tricomp $\mathcal{T}$ shown in Figure 9.2a.

Next we construct a graph $H$ whose face-on-vertex graph has $G'_F$ as a subgraph. Graph $H$ has the required and dummy vertices of $G'_F$ as vertices. It contains an edge $\{x, y\}$ if there exists a face-vertex $f^*$ in $G'_F$ such that edges $\{f^*, x\}$ and $\{f^*, y\}$ appear consecutively in the clockwise order around $f^*$. This construction is shown in Figure 9.4b. The embedding $\hat{H}$ of graph $H$ has a number of faces which are not

represented by face vertices in $G'_F$. (In Figure 9.4b, the only such face is the outer face.) In order to construct $C_\mathcal{T}$, we augment graph $H$ so that the resulting graph $C_\mathcal{T}$ is triconnected and has the property that every face in the unique planar embedding of $C_\mathcal{T}$ which does not correspond to a face vertex in $G'_F$ has at most one required vertex on its boundary. Thus, there exists a natural bijection between the faces of $\hat{\mathcal{T}}$ and $\hat{C}_\mathcal{T}$ with at least two required vertices on their boundaries.

The augmentation of $H$ proceeds in three phases: First we iterate over all faces of $\hat{H}$ which do not correspond to face vertices in $G'_F$. For every boundary cycle of such a face $f$, let $v_1, \ldots, v_k$ be the required vertices in that cycle. (Note that this cycle is not necessarily simple, and some required vertices may appear more than once along the cycle.) For each vertex $v_i$, we split the two edges preceding and succeeding $v_i$ in the cycle by adding two dummy vertices $u_i$ and $w_i$ on these two edges. We connect vertices $u_i$ and $w_i$ by an edge $\{u_i, w_i\}$. This partitions face $f$ into a number of triangles $(u_i, v_i, w_i)$ and a face $f'$ which does not have any required vertex on its boundary. If $H$ is disconnected, the graph obtained after this augmentation is also disconnected. This is equivalent to some face $f'$ having more than one boundary cycle. As long as there is such a face $f'$, we choose two of its boundary cycles and two pairs of consecutive vertices $\{x, y\}$ and $\{u, v\}$ on these two cycles. Assume that $x, y$ and $u, v$ appear in this order clockwise along these two cycles. Then we add edges $\{x, v\}, \{x, u\}, \{y, u\}$ to $H$, thereby concatenating the two boundary cycles. Once this procedure is finished, every face of $H$ has a single boundary cycle. Now we triangulate each face $f'$ not corresponding to a vertex in $G'_F$ by adding a dummy vertex in the center of $f'$ and connecting this vertex to every vertex on the boundary of $f'$. The resulting graph is the constraint graph $C_\mathcal{T}$ of $\mathcal{T}$ (see Figure 9.4c).

**Lemma 9.3** *The constraint graph $C_\mathcal{T}$ of a triconnected component $\mathcal{T}$ is a planar graph with $\mathcal{O}(|R(\mathcal{T})|)$ vertices.*

*Proof.* For bonds and cycles the claim trivially holds, as all vertices in $C_\mathcal{T}$ are required. If $\mathcal{T}$ is a triconnected simple graph, we show the lemma as follows: By Lemma 9.2, graph $G'_F$ contains at most $2|R(\mathcal{T})|$ face vertices of degree one and at most $6|R(\mathcal{T})|$

face vertices of degree two. Thus, at most $10|R(\mathcal{T})|$ dummy vertices are added to $G'_F$ in order to increase the degree of each face vertex to at least three. Hence, graph $H$ is a planar graph with at most $11|R(\mathcal{T})|$ vertices. The construction of $C_\mathcal{T}$ from $H$ adds at most one vertex per edge of $H$ and at most one vertex per face of $H$, $55|R(\mathcal{T})|$ vertices in total. Thus, $C_\mathcal{T}$ has at most $66|R(\mathcal{T})|$ vertices. The planarity of $C_\mathcal{T}$ is explicitly guaranteed by the above construction. □

**Lemma 9.4** *The constraint graph $C_\mathcal{T}$ of a triconnected simple graph $\mathcal{T}$ is a triconnected simple graph.*

*Proof.* It is easily verified that every edge is added at most once to $C_\mathcal{T}$. Hence, $C_\mathcal{T}$ is simple. To show that $C_\mathcal{T}$ is triconnected, let $\hat{C}_\mathcal{T}$ be the planar embedding of $C_\mathcal{T}$ derived from the planar embedding $\hat{\mathcal{T}}$ of $\mathcal{T}$ using the above construction. By Lemma 9.1, it is sufficient to show that for all faces $f_1$ and $f_2$ sharing two vertices $v$ and $w$, edge $\{v, w\}$ is on the boundary of both $f_1$ and $f_2$.

To prove this, we partition the faces of $\hat{C}_\mathcal{T}$ into two categories: *Required* faces are those that correspond to face vertices in $G'_F$. All other faces are *auxiliary* faces.

If $f_1$ and $f_2$ are both required faces, then $v$ and $w$ are required vertices. This is true because dummy vertices are created separately for each face vertex of $G'_F$, so that no two required faces share a dummy vertex. Faces $f_1$ and $f_2$ correspond to two faces $f'_1$ and $f'_2$ of $\mathcal{T}$ sharing vertices $v$ and $w$. Thus, by Lemma 9.1, edge $\{v, w\}$ must be on the boundary of both $f'_1$ and $f'_2$. Edges between required vertices are preserved in $C_\mathcal{T}$.

It is easy to verify that all auxiliary faces of $\hat{C}_\mathcal{T}$ are triangles. Thus, if $f_1$ and $f_2$ are both auxiliary faces, edge $\{v, w\}$ is on the boundary of both $f_1$ and $f_2$.

If $f_1$ and $f_2$ are of different types, assume w.l.o.g. that $f_1$ is required and $f_2$ is an auxiliary face. As $f_2$ is a triangle, edge $\{v, w\}$ is on the boundary of face $f_2$. Thus, it remains to show that edge $\{v, w\}$ is on the boundary of face $f_1$.

Let $f$ be the face of $H$ containing face $f_2$. Face $f$ is split into two types of faces: triangles produced by bridging required vertices on the boundary of $f$ and triangles produced by triangulating the resulting face $f'$. If $f_2$ is of the former type, $f_1$ and $f_2$

can only share vertices $u_i$ and $v_i$ or vertices $v_i$ and $w_i$ because vertices $u_i$ and $w_i$ are on the boundaries of different required faces. Assume w.l.o.g. that $f_1$ and $f_2$ share vertices $u_i$ and $v_i$. Vertex $u_i$ has been inserted on an edge $\{x, v_i\}$ on the boundary of a required face $f_0$, so that $u_i$, $v_i$, and edge $\{u_i, v_i\}$ are on the boundary of face $f_0$. As required faces are not partitioned by our algorithm, and $f_0$ is the only required face having vertex $u_i$ on its boundary, $f_1 = f_0$. Hence, edge $\{u_i, v_i\}$ is on the boundary of face $f_1$.

If $f_2$ is of the latter type, the two vertices shared by $f_1$ and $f_2$ are the endpoints of an edge on the boundary of face $f'$. The edges on the boundary of face $f'$ can be partitioned into two classes: edges $\{u_1, w_1\}, \ldots, \{u_k, w_k\}$ introduced by bridging required vertices $v_1, \ldots, v_k$ on the boundary of face $f$, and edges on the boundaries of required faces. Since no required face can have both endpoints of an edge $\{u_i, w_i\}$ on its boundary, the vertices shared by $f_1$ and $f_2$ must be the endpoints $v$ and $w$ of an edge of the latter type. Both $v$ and $w$ are dummy vertices because no vertex of face $f'$ is required. Hence, there is only one required face which has both vertices on its boundary. As the face which has edge $\{v, w\}$ on its boundary must also have vertices $v$ and $w$ on its boundary, it follows that edge $\{v, w\}$ is on the boundary of face $f_1$. □

**Lemma 9.5** *Let $\hat{\mathcal{T}}$ be a planar embedding of $\mathcal{T}$, and $\hat{C}_\mathcal{T}$ be a planar embedding of $C_\mathcal{T}$. Let $f_1, \ldots, f_k$ be the faces of $\hat{\mathcal{T}}$ with at least two required vertices on their boundaries, and $f'_1, \ldots, f'_l$ be the faces of $\hat{C}_\mathcal{T}$ with at least two required vertices on their boundaries. Then $k = l$, and there exists a bijection $\sigma : [1, k] \to [1, k]$ such that faces $f_i$ and $f'_{\sigma(i)}$ have the same required vertices on their boundaries, in the same order.*

*Proof.* The lemma holds trivially for bonds and simple cycles. If $\mathcal{T}$ is a triconnected simple graph, $C_\mathcal{T}$ is triconnected, by Lemma 9.4. Hence, embeddings $\hat{\mathcal{T}}$ and $\hat{C}_\mathcal{T}$ are unique. In particular, $\hat{C}_\mathcal{T}$ is the embedding of $C_\mathcal{T}$ derived from $\hat{\mathcal{T}}$ by our construction above. Faces $f_1, \ldots, f_k$ in $\hat{\mathcal{T}}$ correspond to face vertices $f^*_1, \ldots, f^*_k$ in $G'_F$, which in

turn correspond to faces $f'_1, \ldots, f'_k$ in $\hat{C}_{\mathcal{T}}$. It is easy to verify that our construction preserves the order of required vertices around these faces.

Thus, in order to prove the lemma, we have to show that each auxiliary face has at most one required vertex on its boundary. Each such face is the result of partitioning a face $f$ of graph $H$ in the above construction which does not correspond to a vertex in $G'_F$. Face $f$ is partitioned into triangles $(u_i, v_i, w_i)$, for each required vertex $v_i$ on the boundary of $f$, and a face $f'$ which does not have any required vertices on its boundary. Vertices $u_i$ and $w_i$ on the boundary of a triangle $(u_i, v_i, w_i)$ are dummy vertices. Face $f'$ is partitioned into triangles none of which has a required vertex on its boundary. Thus, no auxiliary face has more than one required vertex on its boundary. □

Let $\mathcal{T}_1, \ldots, \mathcal{T}_q$ be the tricomps of graphs $G'_1, \ldots, G'_l$. Then we define a sequence of graphs $G^{(1)}_0, \ldots, G^{(1)}_q$ such that $G^{(1)}_0 = G$ and $G^{(1)}_i = G^{(1)}_{i-1}[\mathcal{T}_i^\circ / C^\circ_{\mathcal{T}_i}]$, for $1 \leq i \leq q$. Graph $G^{(1)}$ is defined as $G^{(1)} = G^{(1)}_q$.

**Lemma 9.6** *Graph $G^{(1)}_i$ is planar if and only if $G^{(1)}_{i-1}$ is planar, for $1 \leq i \leq q$. A planar embedding of $G^{(1)}_{i-1}$ can be obtained by locally replacing the embedding of $C^\circ_{\mathcal{T}_i}$ induced by a planar embedding $\hat{G}^{(1)}_i$ of $G^{(1)}_i$ with a consistent embedding of $\mathcal{T}_i^\circ$.*

*Proof.* Consider an embedding $\hat{G}^{(1)}_{i-1}$ of $G^{(1)}_{i-1}$. If $\mathcal{T}_i$ is a bond, then $C_{\mathcal{T}_i} = \mathcal{T}_i$. Hence, $G^{(1)}_{i-1} = G^{(1)}_i$. If $\mathcal{T}_i$ is a cycle, then it is easy to verify that both $\hat{\mathcal{T}}_i^\circ$ and $\hat{C}^\circ_{\mathcal{T}_i}$ have either one or two faces, and that the required vertices on the boundaries of these faces appear in the same order along these boundaries. Hence, any subgraph of $\bar{\mathcal{T}}_i^\circ$ embedded inside a face of $\hat{\mathcal{T}}_i^\circ$ can be embedded without change in the corresponding face of $\hat{C}^\circ_{\mathcal{T}_i}$.

Now consider the case when $\mathcal{T}_i$ is a triconnected simple graph. Let $\hat{\mathcal{T}}_i$ be the unique planar embedding of $\mathcal{T}_i$, and $\hat{\mathcal{T}}_i^\circ$ be the planar embedding of $\mathcal{T}_i^\circ$ obtained from $\hat{\mathcal{T}}_i$ by removing all virtual edges in $\mathcal{T}_i$. Let $\hat{C}_{\mathcal{T}_i}$ be the unique planar embedding of $C_{\mathcal{T}_i}$, and $\hat{C}^\circ_{\mathcal{T}_i}$ be the planar embedding of $C^\circ_{\mathcal{T}_i}$ obtained from $\hat{C}_{\mathcal{T}_i}$ by removing all virtual edges in $C_{\mathcal{T}_i}$.

We partition $\bar{\mathcal{T}}_i^\circ$ into maximal subgraphs each of which is embedded inside a face $f$ of $\hat{\mathcal{T}}_i^\circ$. Each such graph $K$ is incident only to required vertices on the boundary of $f$. If $f$ is the result of merging a number of faces of $\hat{\mathcal{T}}_i$ by removing virtual edges, each constituent face of $f$ in $\hat{\mathcal{T}}_i$ has at least two required vertices on its boundary. Thus, these constituent faces are preserved in $\hat{C}_{\mathcal{T}_i}$, by Lemma 9.5. Moreover, they share the same virtual edges in $\hat{C}_{\mathcal{T}_i}$ as in $\hat{\mathcal{T}}_i$. Thus, face $f$ has a corresponding face $f'$ in $\hat{C}^\circ_{\mathcal{T}_i}$ with the same required vertices on its boundary, in the same order. If $f$ is a face of $\hat{\mathcal{T}}_i^\circ$ consisting of a single face of $\hat{\mathcal{T}}_i$ with at least two required vertices on its boundary, this face is preserved in $\hat{C}_{\mathcal{T}_i}$ and thus in $\hat{C}^\circ_{\mathcal{T}_i}$, by Lemma 9.5. In both cases, $K$ can be embedded inside $f'$ without changing the embedding of $K$. If face $f$ has only one required vertex on its boundary, $K$ can be embedded inside an arbitrary face of $\hat{C}^\circ_{\mathcal{T}_i}$ with this vertex on its boundary. Embedding all subgraphs $K$ in this manner results in a planar embedding of $G_i^{(1)}$. The proof that $G_{i-1}^{(1)}$ is planar if $G_i^{(1)}$ is planar is similar. □

**Corollary 9.1** *Graph $G^{(1)}$ is planar if and only if graph $G$ is planar. A planar embedding of $G$ can be obtained from an embedding $\hat{G}^{(1)}$ of $G^{(1)}$ by locally replacing the embeddings of graphs $C^\circ_{\mathcal{T}_1}, \ldots, C^\circ_{\mathcal{T}_q}$ with consistent embeddings of graphs $\mathcal{T}_1^\circ, \ldots, \mathcal{T}_q^\circ$.*

## 9.5 The Constraint Graph of a Bicomp

Given graph $G^{(1)}$ obtained by replacing each tricomp of graphs $G'_1, \ldots, G'_l$ with its constraint graph, we show in this section how to construct the constraint graph of a bicomp $\mathcal{B}$. In order to do this, we use a two-step procedure to classify the tricomps of $\mathcal{B}$ as essential or inessential. For an essential tricomp $\mathcal{T}$ of $\mathcal{B}$, the constraint graph $C_{\mathcal{T}}$ of $\mathcal{T}$ in $G^{(1)}$ is left unchanged. An inessential tricomp is either completely removed from $G^{(1)}$, or it is grouped together with other inessential tricomps. Each such group is then replaced with a constraint graph of constant size.

Let $R(\mathcal{B})$ be the set of required vertices of $\mathcal{B}$, and $\mathcal{T}_1, \ldots, \mathcal{T}_q$ be its tricomps. Let $T = T_{\text{tri}}(\mathcal{B})$ be the tricomp tree of $\mathcal{B}$ (see Figure 9.5). Tree $T$ has $q$ vertices

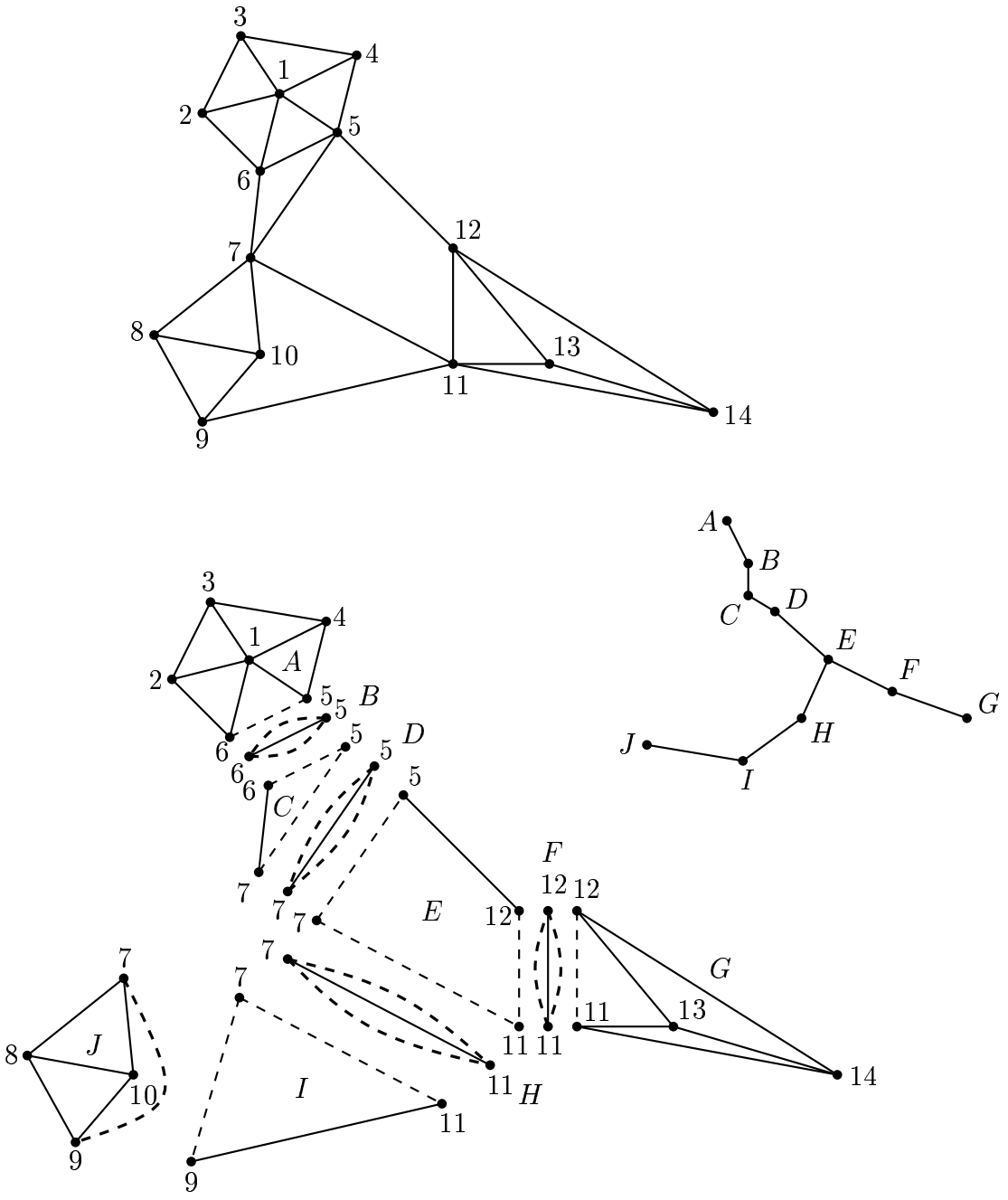



**Figure 9.5**

A biconnected planar graph $G$, its tricomps, and its tricomp tree. Tricomps are labelled with capital letters. Virtual edges in the tricomps are dashed.

$\tau_1, \ldots, \tau_q$, one per tricomp $\mathcal{T}_i$. For each vertex $v \in R(\mathcal{B})$, we choose a tricomp $\mathcal{T}(v)$ such that $v \in \mathcal{T}(v)$. We call a tricomp $\mathcal{T}_i$ *essential* if there is a vertex $v \in R(\mathcal{B})$ such that $\mathcal{T}_i = \mathcal{T}(v)$. A tricomp $\mathcal{T}_j$ is *potentially essential* if there are two essential tricomps $\mathcal{T}_i$ and $\mathcal{T}_k$ such that vertex $\tau_j$ is on the path from $\tau_i$ to $\tau_k$ in $T$. All other tricomps are *inessential*. In the next section, we show that removing all inessential tricomps from $G^{(1)}$ does not alter its (non-)planarity. Then we finish the classification of the tricomps by deciding which of the potentially essential tricomps are essential or inessential. In Section 9.5.2, all tricomps classified as inessential in this second round of classification are replaced by a small number of constraint graphs of constant size. Section 9.5.3 puts the pieces together and shows that the final constraint graph $C_{\mathcal{B}}$ of $\mathcal{B}$ has size $\mathcal{O}(|R(\mathcal{B})|)$.

### 9.5.1 Discarding Inessential Tricomps

Let $T'$ be the tree obtained by removing all vertices $\tau_j$ corresponding to inessential tricomps $\mathcal{T}_j$ from $T$. Let $\mathcal{B}'$ be the subgraph of $\mathcal{B}$ obtained by merging all tricomps corresponding to vertices in $T'$. The nodes $\tau_j \in T$ corresponding to inessential tricomps $\mathcal{T}_j$ induce a set of maximal subtrees $T_1, \ldots, T_s$ of $T$. Each such subtree $T_j$ is connected to $T'$ through a single edge. Let $K_j$ be the subgraph of $\mathcal{B}$ obtained by merging all tricomps corresponding to the nodes in tree $T_j$. As $T_j$ and $T'$ share only a single edge, $K_j$ and $\mathcal{B}'$ share exactly one virtual edge $(v_j, w_j, i_j)$. Let $\mathcal{B}''$ be the graph obtained by replacing graphs $K_1^\circ, \ldots, K_s^\circ$ with edges $(v_1, w_1, i_1), \ldots, (v_s, w_s, i_s)$ in $\mathcal{B}$. Alternatively, $\mathcal{B}''$ is obtained by making all virtual edges in $\mathcal{B}'$ non-virtual.

Let $\mathcal{B}_1, \ldots, \mathcal{B}_q$ be the bicomps of graphs $G'_1, \ldots, G'_l$. Then we define a sequence of graphs $G_0^{(2)}, \ldots, G_q^{(2)}$, where $G_0^{(2)} = G^{(1)}$ and $G_i^{(2)} = G_{i-1}^{(2)}[\mathcal{B}_i/\mathcal{B}''_i]$. Graph $G^{(2)}$ is defined as $G^{(2)} = G_q^{(2)}$.

**Lemma 9.7** *Graph $G_i^{(2)}$ is planar if and only if graph $G_{i-1}^{(2)}$ is planar, for $1 \leq i \leq q$. A planar embedding of $G_{i-1}^{(2)}$ can be obtained by locally replacing edges in an embedding of $G_i^{(2)}$ with embeddings of inessential tricomps of $\mathcal{B}_i$.*

*Proof.* Let $\mathcal{B} = \mathcal{B}_i$, and let trees $T$, $T'$, and $T_1, \ldots, T_s$ and graphs $\mathcal{B}'$ and $K_1, \ldots, K_s$ be defined as above. Let $\hat{G}^{(2)}_{i-1}$ be a planar embedding of graph $G^{(2)}_{i-1}$. For each subgraph $K_j$ of $\mathcal{B}$, there is a path from $v_j$ to $w_j$ in $K^\circ_j$. Thus, replacing $K^\circ_j$ by edge $(v_j, w_j, i_j)$ in $\hat{G}^{(2)}_{i-1}$ corresponds to removing the whole graph $K^\circ_j$ except that path from $G^{(2)}_{i-1}$, and then replacing this path by a single edge. As all tricomps in $K_j$ are inessential, $v_j$ and $w_j$ are the only vertices shared by $K^\circ_j$ and $\bar{K}^\circ_j$. Hence, two paths from $v_j$ to $w_j$ in graph $K^\circ_j$ and from $v_k$ to $w_k$ in another graph $K^\circ_k$ are internally vertex disjoint, so that replacing both paths by a single edge does not introduce an intersection in the planar embedding of $G^{(2)}_{i-1}$. Applying this argument to graphs $K^\circ_1, \ldots, K^\circ_s$ in turn shows that $G^{(2)}_i$ is planar if $G^{(2)}_{i-1}$ is planar.

To see that $G^{(2)}_{i-1}$ is planar if $G^{(2)}_i$ is planar, recall that each graph $K^\circ_j$ shares only vertices $v_j$ and $w_j$ with $\bar{K}^\circ_j$. Hence, an embedding of $G^{(2)}_{i-1}$ can be obtained from an embedding of $G^{(2)}_i$ by replacing each edge $(v_j, w_j, i_j)$ in $G^{(2)}_i$ with an embedding of graph $K^\circ_j$ which has vertices $v_j$ and $w_j$ on its outer face. The existence of such an embedding follows immediately from the planarity of the tricomps of a biconnected planar graph. □

**Corollary 9.2** *Graph $G^{(2)}$ is planar if and only if graph $G^{(1)}$ is planar. A planar embedding of $G^{(1)}$ can be obtained by locally replacing edges in an embedding of $G^{(2)}$ with embeddings of inessential tricomps.*

Having disposed of the first set of inessential tricomps, we now classify the potentially essential tricomps of $\mathcal{B}''$ as essential or inessential. A potentially essential tricomp is essential if its corresponding vertex in $T'$ has degree at least three. Otherwise, it is inessential. Note that all vertices in $T'$ whose corresponding tricomps are inessential as a result of this second classification have degree two. This is true because all leaves correspond to essential tricomps, and all tricomps corrsponding to internal nodes of degree at least three have been declared essential. We partition the set of nodes corresponding to inessential tricomps into maximal paths in $T'$.

Let $P = (\tau_1, \ldots, \tau_s)$ be such a path, and let $\tau_0$ and $\tau_{s+1}$ be the two other neighbors of $\tau_1$ and $\tau_s$, respectively. Tricomps $\mathcal{T}_0$ and $\mathcal{T}_{s+1}$ are essential. Let $(v_j, w_j, i_j)$ be

the virtual edge shared by tricomps $\mathcal{T}_j$ and $\mathcal{T}_{j+1}$, for $0 \leq j \leq s$. Let $H_P$ be the graph obtained by merging tricomps $\mathcal{T}_1, \ldots, \mathcal{T}_s$. Then $H_P$ contains two virtual edges: $(v_0, w_0, i_0)$ and $(v_s, w_s, i_s)$. Also, as all tricomps in $H_P$ are inessential, graph $H_P^\circ$ shares only vertices $v_0$, $w_0$, $v_s$, and $w_s$ with $\bar{H}_P^\circ$. Next we construct a constraint graph $C_P$ of constant size for each such graph $H_P$ and replace $H_P^\circ$ with $C_P^\circ$.

### 9.5.2 Compressing Chains of Inessential Tricomps

To construct the constraint graph $C_P$ for a graph $H_P$, we partition the tricomps $\mathcal{T}_1, \ldots, \mathcal{T}_s$ of $H_P$ into five (possibly empty) groups, and replace each such group by its own constraint graph of constant size. These five groups are defined as follows:

Let $j_0$ be the maximal index such that $\{v_{j_0}, w_{j_0}\} \cap \{v_0, w_0\} \neq \emptyset$. Then the *fan* of tricomp $\mathcal{T}_0$ is the union of tricomps $\mathcal{T}_1, \ldots, \mathcal{T}_{j_0}$. The fan of $\mathcal{T}_0$ is empty if $j_0 = 0$.

The fan of tricomp $\mathcal{T}_{s+1}$ is defined analogously: Let $j_s$ be the minimal index such that $\{v_{j_s}, w_{j_s}\} \cap \{v_s, w_s\} \neq \emptyset$. Then the *fan* of tricomp $\mathcal{T}_{s+1}$ is the union of tricomps $\mathcal{T}_{j_s+1}, \ldots, \mathcal{T}_s$. The fan of $\mathcal{T}_{s+1}$ is empty if $j_s = s$.

If $j_0 < j_s$, let $\mathcal{T}_{j_0+1}$ be the *separating tricomp* for $\mathcal{T}_0$. If $j_0 < j_s - 1$, let $\mathcal{T}_{j_s}$ be the separating tricomp for $\mathcal{T}_{s+1}$. If $j_0 < j_s - 2$, let the union of tricomps $\mathcal{T}_{j_0+2}, \ldots, \mathcal{T}_{j_s-1}$ be the *core* of graph $H_P$. Figure 9.6 illustrates these definitions. Each non-empty fan, separating tricomp, or core is replaced by its own constraint graph of constant size. For a separating tricomp $\mathcal{T}$, we keep the constraint graph $C_\mathcal{T}$ constructed in Section 9.4. The constraint graphs of fans and cores are described next.

#### 9.5.2.1 The Constraint Graph of a Fan

Let $\mathcal{F}$ be a fan of some graph $H_P$ with virtual edges $(a, b, i)$ and $(a, c, j)$. For a non-empty fan $\mathcal{F}$, we distinguish two cases. If $b = c$, the fan consists of a single bond. In this case, the constraint graph of $\mathcal{F}$ is the constraint graph of the bond, which is the bond itself. Otherwise, the constraint graph has to capture the possible embeddings of vertices $a$, $b$, and $c$ and virtual edges $(a, b, i)$ and $(a, c, j)$. In particular, we have to distinguish the following cases (see Figure 9.7):

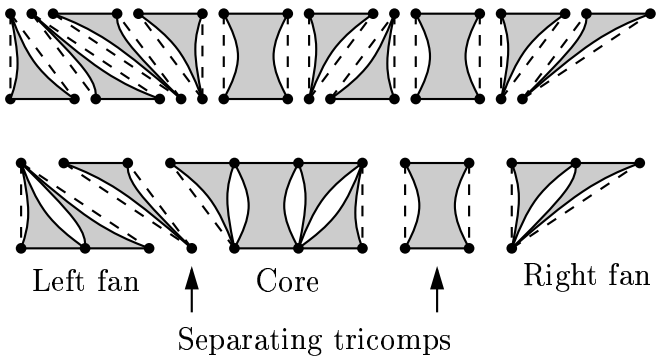



**Figure 9.6**
Tricomps $\mathcal{T}_1, \ldots, \mathcal{T}_s$ and the five graphs into which the merge of these tricomps is being decomposed.

(a) Fan $\mathcal{F}$ has a planar embedding $\hat{\mathcal{F}}$ such that there are two faces which have edges $(a, b, i)$ and $(a, c, j)$ on their boundaries.

(b) Fan $\mathcal{F}$ has a planar embedding $\hat{\mathcal{F}}$ such that there is one face which has edges $(a, b, i)$ and $(a, c, j)$ on its boundary, and another face with edge $(a, b, i)$ and vertex $c$ on its boundary. (There is a symmetric case where the second face has edge $(a, c, j)$ and vertex $b$ on its boundary.)

(c) Fan $\mathcal{F}$ has a planar embedding $\hat{\mathcal{F}}$ such that there is one face which has edges $(a, b, i)$ and $(a, c, j)$ on its boundary, and another face with vertices $a$, $b$, and $c$ on its boundary.

(d) Fan $\mathcal{F}$ has a planar embedding $\hat{\mathcal{F}}$ such that there is one face which has edges $(a, b, i)$ and $(a, c, j)$ on its boundary.

(e) Fan $\mathcal{F}$ has a planar embedding $\hat{\mathcal{F}}$ such that there is at least one face which has vertices $b$ and $c$ on its boundary.

(f) For every face of any embedding $\hat{\mathcal{F}}$ of $\mathcal{F}$, the required vertices on its boundary are either in $\{a, b\}$ or in $\{a, c\}$.

It is easy to verify that if fan $\mathcal{F}$ satisfies one of the above conditions, then its constraint graph $C_{\mathcal{F}}$ shown in Figure 9.7 satisfies the same condition, and vice versa.

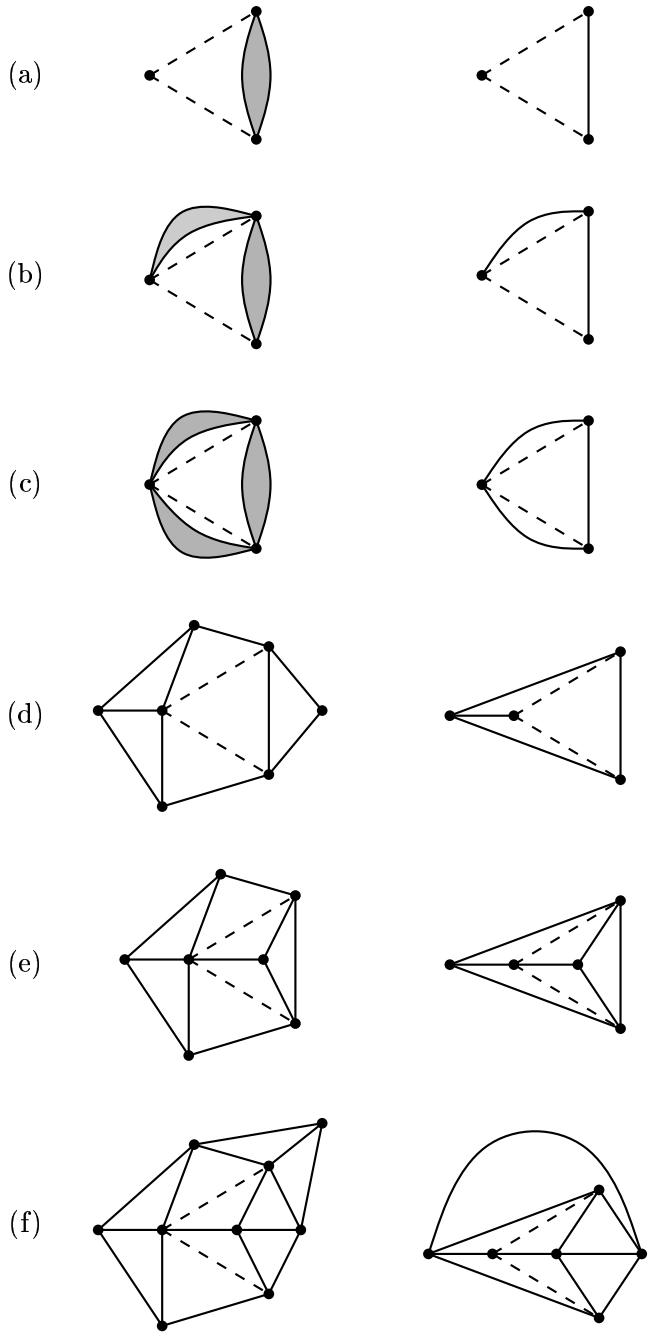



**Figure 9.7**
Fans illustrating the different possible constellations of virtual edges $(a, b, i)$ and $(a, c, j)$. The left graph in each figure is a fan $\mathcal{F}$. The right graph is its constraint graph $C_{\mathcal{F}}$.

Observe that there is always a face with vertices $a$ and $b$ on its boundary, and a face with vertices $a$ and $c$ on its boundary, as $\mathcal{F}$ contains edges $(a, b, i)$ and $(a, c, j)$. Thus, the distinction is whether there is a face with vertices $b$ and $c$ on its boundary. In Cases (a)–(e), such a face exists. In Case (f), such a face does not exist. Cases (a)–(e) further distinguish whether there are faces that have all three vertices on their boundaries. In Cases (a)–(d), such a face exists. In Case (e), it does not. Now observe that if there is a face with vertices $a$, $b$, and $c$ on its boundary, it can be ensured that it has edges $(a, b, i)$ and $(a, c, j)$ on its boundary, by embedding these two edges inside that face. Thus, the only difference between Cases (a)–(d) is whether there is a second face with all three vertices on its boundary and whether this second face has none, one, or both of the virtual edges on its boundary. Hence, Cases (a)–(f) are the only possibilities for the structure of fan $\mathcal{F}$.

It is easy to test which of the six cases applies: To test for Case (a), add two extra vertices $x$ and $y$ on the two virtual edges and connect each of the vertices $a$, $b$, $c$, $x$, and $y$ to two vertices $z_1$ and $z_2$ representing the two faces in Case (a). Case (a) applies if and only if the resulting graph is planar. To test for Case (b), we remove edge $\{y, z_2\}$ and test for planarity again. (In order to test for the symmetric case, remove edge $\{x, z_2\}$ instead of edge $\{y, z_2\}$.) To test for Case (c), we remove both edges $\{x, z_2\}$ and $\{y, z_2\}$. To test for Case (d), we remove vertex $z_2$ and all incident edges. To test for Case (e), we remove edges $\{a, z_1\}$, $\{x, z_1\}$, and $\{y, z_1\}$. If none of these graphs is planar, Case (f) applies.

Let $\mathcal{F}_1, \ldots, \mathcal{F}_q$ be the fans of all graphs $H_P$ in $G^{(2)}$. Then we define a sequence of graphs $G_0^{(3)}, \ldots, G_q^{(3)}$, where $G_0^{(3)} = G^{(2)}$ and $G_i^{(3)} = G_{i-1}^{(3)}[\mathcal{F}_i^\circ / C_{\mathcal{F}_i}^\circ]$, for $1 \leq i \leq q$. Graph $G^{(3)}$ is defined as $G^{(3)} = G_q^{(3)}$.

**Lemma 9.8** *Graph $G_i^{(3)}$ is planar if and only if $G_{i-1}^{(3)}$ is planar, for $1 \leq i \leq q$. A planar embedding of $G_{i-1}^{(3)}$ can be obtained from a planar embedding of $G_i^{(3)}$ by locally replacing the embedding of $C_{\mathcal{F}_i}^\circ$ with a consistent embedding of $\mathcal{F}_i^\circ$.*

*Proof.* Consider a planar embedding $\hat{G}_{i-1}^{(3)}$ of graph $G_{i-1}^{(3)}$. Let $\hat{\mathcal{F}}_i^\circ$ be the planar embedding of $\mathcal{F}_i^\circ$ induced by $\hat{G}_{i-1}^{(3)}$. We partition $\bar{\mathcal{F}}_i^\circ$ into maximal subgraphs so that

each such subgraph $K$ is embedded inside a face of $\hat{\mathcal{F}}_i^\circ$. We denote the two virtual edges in $\mathcal{F}_i$ by $(a, b, j)$ and $(a, c, k)$.

If $\hat{\mathcal{F}}_i$ has at least one face with vertices $a$, $b$, and $c$ on its boundary, one of Cases (a)–(d) applies. It is easy to verify that in all three cases, the embeddings $\hat{\mathcal{F}}_i^\circ$ and $\hat{C}_{\mathcal{F}_i}^\circ$ as shown in Figure 9.7 have the same faces with all three vertices on their boundaries. Thus, a graph $K$ embedded inside a face of $\hat{\mathcal{F}}_i^\circ$ with all three vertices $a$, $b$, and $c$ on its boundary can be embedded in the corresponding face of $\hat{C}_{\mathcal{F}_i}^\circ$.

It is easy to verify that in all six cases, for every face $f$ in $\hat{\mathcal{F}}_i^\circ$ with two required vertices $v$ and $w$ on its boundary, there exists a face $f'$ in $\hat{C}_{\mathcal{F}_i}^\circ$ so that vertices $v$ and $w$ appear consecutively[1] on the boundary of face $f'$. Hence, any graph $K$ embedded in such a face $f$ can be embedded inside the corresponding face $f'$ without intersecting any of the other graphs possibly embedded inside face $f'$.

Any graph $K$ embedded in a face $f$ of $\hat{\mathcal{F}}_i^\circ$ which has only one required vertex on its boundary can be embedded inside any face of $\hat{C}_{\mathcal{F}_i}^\circ$ which has the same required vertex on its boundary, without creating any conflicts.

Thus, by embedding all subgraphs $K$ of $\bar{\mathcal{F}}_i^\circ$ in this manner, a planar embedding $\hat{G}_i^{(3)}$ of graph $G_i^{(3)}$ is obtained from the given embedding $\hat{G}_{i-1}^{(3)}$ of graph $G_{i-1}^{(3)}$.

Now assume that a planar embedding $\hat{G}_i^{(3)}$ of $G_i^{(3)}$ is given. Let $\hat{C}_{\mathcal{F}_i}$ be the embedding of $C_{\mathcal{F}_i}$ induced by $\hat{G}_i^{(3)}$, and let $\hat{C}_{\mathcal{F}_i}^\circ$ be the restriction of $\hat{C}_{\mathcal{F}_i}$ to $C_{\mathcal{F}_i}^\circ$. It is easy to verify that in each of the above cases, there exists an embedding $\hat{\mathcal{F}}_i$ of $\mathcal{F}_i$ such that for every face of $\hat{C}_{\mathcal{F}_i}^\circ$, there exists a corresponding face in the restriction $\hat{\mathcal{F}}_i^\circ$ of $\hat{\mathcal{F}}_i$ to $\mathcal{F}_i^\circ$ with the same required vertices on its boundary. Moreover, if there are two faces in $\hat{C}_{\mathcal{F}_i}^\circ$ with three required vertices on their boundaries, then there are two such faces in $\hat{\mathcal{F}}_i^\circ$. Using the same arguments as above, this implies that $G_{i-1}^{(3)}$ is planar if $G_i^{(3)}$ is planar. □

[1]Here, two required vertices are said to be consecutive if there exists no *required* vertex between them on the boundary of the face. There may be other vertices between them.

**Corollary 9.3** *Graph $G^{(3)}$ is planar if and only if $G^{(2)}$ is planar. A planar embedding of $G^{(2)}$ can be obtained from a planar embedding of $G^{(3)}$ by locally replacing the embeddings of graphs $C^{\circ}_{\mathcal{F}_1}, \ldots, C^{\circ}_{\mathcal{F}_q}$ with consistent embeddings of graphs $\mathcal{F}^{\circ}_1, \ldots, \mathcal{F}^{\circ}_q$.*

#### 9.5.2.2 The Constraint Graph of a Core

For the core $\mathcal{C}$ of a graph $H_P$, its constraint graph has to capture the different embeddings of its two virtual edges $(a, b, i)$ and $(c, d, j)$. If $\{a, b\} = \{c, d\}$, $\mathcal{C}$ consists of a single bond, and we define $C_{\mathcal{C}} = \mathcal{C}$. So assume that $\{a, b\} \neq \{c, d\}$. Then there are three possibilities (see Figure 9.8):

(a) There exists an embedding of $\mathcal{C}$ which has two faces with edges $(a, b, i)$ and $(c, d, j)$ on their boundaries.

(b) There exists an embedding of $\mathcal{C}$ which has one face with edges $(a, b, i)$ and $(c, d, j)$ on its boundary.

(c) There exists no embedding of $\mathcal{C}$ such that edges $(a, b, i)$ and $(c, d, j)$ appear on the same face.

It is easy to verify that if core $\mathcal{C}$ satisfies one of the above conditions, then its constraint graph $C_{\mathcal{C}}$ shown in Figure 9.8 satisfies the same condition, and vice versa. Moreover, these are the only possibilities for the structure of core $\mathcal{C}$, as there cannot be three faces with edges $(a, b, j)$ and $(c, d, k)$ on their boundaries.

These three possibilities can be tested in a way similar to the processing of a fan: We split edge $(a, b, i)$ into two edges $\{a, x\}$ and $\{x, b\}$ and edge $(c, d, j)$ into two edges $\{c, y\}$ and $\{y, d\}$, add two vertices $z_1$ and $z_2$ to $\mathcal{C}$, and connect both of them to vertices $a$, $b$, $c$, $d$, $x$, and $y$. The first case applies if and only if the resulting graph is planar. To test for the second case, we remove vertex $z_2$ and its incident edges from the graph and test for planarity again. If both tests fail, the third case applies.

Let $\mathcal{C}_1, \ldots, \mathcal{C}_q$ be the cores of all graphs $H_P$ in $G^{(3)}$. Then we define a sequence of graphs $G^{(4)}_0, \ldots, G^{(4)}_q$, where $G^{(4)}_0 = G^{(3)}$ and $G^{(4)}_i = G^{(4)}_{i-1}[\mathcal{C}^{\circ}_i / C^{\circ}_{\mathcal{C}_i}]$, for $1 \leq i \leq q$. Graph $G^{(4)}$ is defined as $G^{(4)} = G^{(4)}_q$.

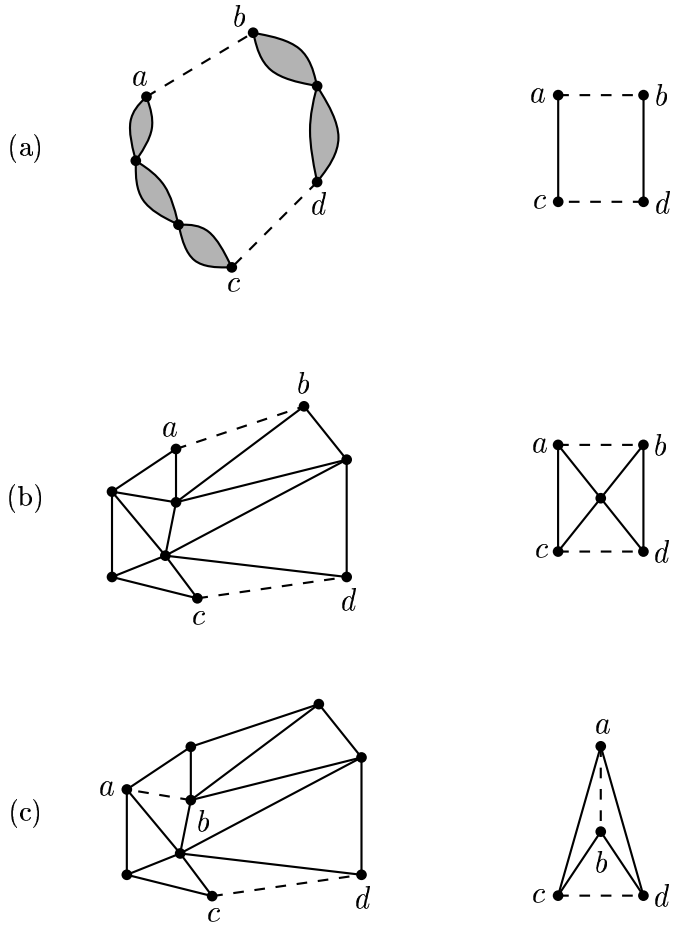



**Figure 9.8**
Three examples of cores illustrating the different possible constellations of virtual edges $(a, b, j)$ and $(c, d, k)$. The left graph in each figure is a core $\mathcal{C}$. The right graph is its constraint graph $C_{\mathcal{C}}$.

**Lemma 9.9** *Graph $G_i^{(4)}$ is planar if and only if graph $G_{i-1}^{(4)}$ is planar, for $1 \leq i \leq q$. A planar embedding of $G_{i-1}^{(4)}$ can be obtained from a planar embedding of $G_i^{(4)}$ by replacing the embedding of $C_{\mathcal{C}_i}^{\circ}$ induced by the embedding of $G_i^{(4)}$ with a consistent embedding of $\mathcal{C}_i^{\circ}$.*

*Proof.* Let $\hat{G}_{i-1}^{(4)}$ be a planar embedding of $G_{i-1}^{(4)}$, and let $\hat{\mathcal{C}}_i^{\circ}$ be the embedding of $\mathcal{C}_i^{\circ}$ induced by $\hat{G}_{i-1}^{(4)}$. Splits $s(a, b, j)$ and $s(c, d, k)$ partition $\mathcal{B}$ into three graphs $\mathcal{C}_i$, $K_1$,

and $K_2$. Graph $K_1$ shares virtual edge $(a, b, j)$ with $\mathcal{C}_i$. Graph $K_2$ shares virtual edge $(c, d, k)$ with $\mathcal{C}_i$. By the construction of core $\mathcal{C}_i$, none of the vertices in $\mathcal{C}_i$ is required in the bicomp $\mathcal{B}$ containing $\mathcal{C}_i$. Hence, no edge in $\bar{\mathcal{B}}$ can be incident to any vertex in $\mathcal{C}_i$. That is, all components of $\bar{\mathcal{B}}$ are adjacent to either $K_1$ or $K_2$. In particular, $\bar{\mathcal{C}}_i^\circ$ can be split into at most two subgraphs $G_1$ and $G_2$ embedded inside the faces of $\hat{\mathcal{C}}_i^\circ$ containing edges $(a, b, j)$ and $(c, d, k)$.

It is easy to verify that in all three cases depicted in Figure 9.8, the order of required vertices along a face $f$ of $\hat{\mathcal{C}}_i$ containing one of the graphs $G_1$ or $G_2$ is preserved in the corresponding face $f'$ of $\hat{C}_{\mathcal{C}_i}$. Hence, this graph can be embedded in $f'$ without changing its embedding, and we obtain a planar embedding of $G_i^{(4)}$.

In order to show that $G_{i-1}^{(4)}$ is planar if $G_i^{(4)}$ is planar, we reverse the above argument. The only case that deserves closer attention is Case (c) in Figure 9.8. In this case, $\hat{C}_{\mathcal{C}_i}^\circ$ contains two faces $(a, c, b, d)$; but the corresponding faces of $\hat{\mathcal{C}}_i^\circ$ have only vertices $a$ and $b$ or $c$ and $d$ on their boundaries. Given a planar embedding $\hat{G}_i^{(4)}$ of $G_i^{(4)}$, we have to show that every subgraph of $\bar{\mathcal{C}}_i^\circ$ embedded completely inside one of the faces of $\hat{C}_{\mathcal{C}_i}^\circ$ contains only vertices $a$ and $b$ or $c$ and $d$. Let $K$ be such a subgraph embedded inside a face $f$ of $\hat{C}_{\mathcal{C}_i}^\circ$, and assume for the sake of contradiction that $K$ contains vertices $a$, $b$, and $c$. Since no vertex in $\mathcal{C}_i$ is required, graph $K$ contains both graphs $K_1$ and $K_2$ defined above. However, graphs $K_1$ and $K_2$ contain two disjoint paths from $a$ to $b$ and from $c$ to $d$. Since vertices, $a, c, b, d$ appear in this order along the boundary of face $f$, these two paths intersect, contradicting the fact that $\hat{G}_i^{(4)}$ is a planar embedding of $G_i^{(4)}$. □

**Corollary 9.4** *Graph $G^{(4)}$ is planar if and only if graph $G^{(3)}$ is planar. A planar embedding of $G^{(3)}$ can be obtained from a planar embedding of $G^{(4)}$ by replacing the embeddings of graphs $C_{\mathcal{C}_1}^\circ, \ldots, C_{\mathcal{C}_q}^\circ$ with consistent embeddings of graphs $\mathcal{C}_1^\circ, \ldots, \mathcal{C}_q^\circ$.*

### 9.5.3 The Constraint Graph of the Bicomp

The above construction replaces every bicomp $\mathcal{B}$ in $G$ with a multigraph $C'_{\mathcal{B}}$. In order to finish the construction, we remove all multiple edges from $G^{(4)}$. Let $G^{(5)}$

be the resulting graph. The construction of $G^{(5)}$ is equivalent to the following two-step procedure: First we replace every bicomp $\mathcal{B}$ with a graph $C_{\mathcal{B}}$, which is obtained from $C'_{\mathcal{B}}$ by removing multiple edges. Then we remove remaining multiple edges from the union of graphs $C_{\mathcal{B}_1}, \ldots, C_{\mathcal{B}_q}$, where $\mathcal{B}_1, \ldots, \mathcal{B}_q$ are the bicomps of graphs $G'_1, \ldots, G'_l$. Graph $C_{\mathcal{B}}$ is the constraint graph of bicomp $\mathcal{B}$. The following lemma is obvious.

**Lemma 9.10** *Graph $G^{(5)}$ is planar if and only if $G^{(4)}$ is planar. A planar embedding of $G^{(4)}$ can be obtained from a planar embedding $\hat{G}^{(5)}$ of $G^{(5)}$ by duplicating edges in $\hat{G}^{(5)}$.*

The next lemma shows that the constraint graph $C_{\mathcal{B}}$ of a bicomp $\mathcal{B}$ is small.

**Lemma 9.11** *The constraint graph $C_{\mathcal{B}}$ of a bicomp $\mathcal{B}$ is a simple planar graph with $\mathcal{O}(|R(\mathcal{B})|)$ vertices.*

*Proof.* The planarity of $C_{\mathcal{B}}$ follows immediately from the above construction. We show that there are at most $2|R(\mathcal{B})|$ essential tricomps in $\mathcal{B}$. There are two types of essential tricomps. Type-I tricomps are tricomps $\mathcal{T}(v)$, $v \in R(\mathcal{B})$. Type-II tricomps are tricomps whose corresponding vertices in $T'$ have degree at least three, where $T'$ is the tree constructed from the tricomp tree $T = T_{\text{tri}}(\mathcal{B})$ in Section 9.5.1. Clearly, there are at most $|R(\mathcal{B})|$ tricomps of type I. Let $T''$ be the tree obtained from $T'$ by replacing every maximal path whose internal vertices correspond to inessential tricomps with a single edge. Tree $T''$ contains all vertices of $T'$ corresponding to essential tricomps. All leaves of $T''$ correspond to type-I tricomps, so that there are at most $|R(\mathcal{B})|$ leaves in $T''$. The vertices corresponding to type-II tricomps are a subset of the vertices of degree at least three in $T''$. There can be at most $|R(\mathcal{B})|-2$ such vertices, as there are at most $|R(\mathcal{B})|$ leaves in $T''$. Thus, there are at most $2|R(\mathcal{B})|-2$ essential tricomps in $\mathcal{B}$.

Every edge in $T''$ represents a (possibly empty) path of vertices of degree two in $T'$, which correspond to inessential tricomps. For each such path $P$, the graph $H_P$ obtained by merging the tricomps corresponding to the vertices in $P$ has been replaced

by a constraint graph $C_P$ of constant size. This implies that the total size of all constraint graphs not corresponding to essential tricomps is $\mathcal{O}(|R(\mathcal{B})|)$. An essential tricomp contains at most twice as many vertices belonging to separation pairs as there are edges incident to the corresponding vertex in $T'$. Thus, the total number of required vertices in all essential tricomps is $\mathcal{O}(|R(\mathcal{B})|)$. By Lemma 9.3, this implies that the constraint graphs of these tricomps have total size $\mathcal{O}(|R(\mathcal{B})|)$. As merging all constraint graphs can only reduce the number of vertices in the resulting graph, graph $C'_{\mathcal{B}}$ has $\mathcal{O}(|R(\mathcal{B})|)$ vertices. Graph $C_{\mathcal{B}}$ is obtained from $C'_{\mathcal{B}}$ by removing edges. □

## 9.6 The Constraint Graph of a Connected Component

So far every bicomp $\mathcal{B}$ of graphs $G'_1, \ldots, G'_l$ has been replaced by a small constraint graph $C_{\mathcal{B}}$. In order to obtain constraint graphs $C_1, \ldots, C_l$, we now remove some inessential bicomps altogether. The remaining chains of inessential bicomps are replaced by constraint graphs of constant size. The construction is similar to the construction of the constraint graph $C_{\mathcal{B}}$ of a bicomp $\mathcal{B}$ from the constraint graphs of its tricomps. That is, first the bicomps of a graph $G'_j$ are classified as essential, potentially essential, or inessential. Inessential bicomps are removed. Then we finish the classification of potentially essential bicomps based on the degree of their corresponding vertices in the bicomp-cutpoint-tree of $G'_j$. The remaining inessential bicomps form chains in $G'_j$ sharing only two vertices with the rest of $G'_j$. Each such chain is replaced with a constraint graph of constant size. Next we describe this construction in detail.

Let $G'_j$ be one of the graphs $G'_1, \ldots, G'_l$, let $S_j$ be the set of separator vertices in $G'_j$, and let $\mathcal{B}_1, \ldots, \mathcal{B}_q$ be the bicomps of $G'_j$. Let $T = T_{\text{bic}}(G'_j)$ be the bicomp-cutpoint-tree of $G'_j$. Tree $T$ contains all cutpoints $v_1, \ldots, v_r$ of $G'_j$ and one *bicomp node* $\beta_i$ per bicomp $\mathcal{B}_i$ of $G'_j$. For every vertex $v \in S_j$, we choose a bicomp $\mathcal{B}(v)$ such that $v \in \mathcal{B}(v)$. As $T$ contains bicomp nodes as well as cutpoints, we classify the nodes of $T$ rather than the bicomps of $G'_j$ as essential, potentially essential, or inessential.

A node $\beta_i$ in $T$ is *essential* if there exists a vertex $v \in S_j$ such that $\mathcal{B}(v) = \mathcal{B}_i$. A node $v$ is *potentially essential* if there are two essential nodes $u$ and $w$ in $T$ such that $v$ is on the path from $u$ to $w$ in $T$. All other nodes of $T$ are *inessential*.

In the next section, we show that removing all bicomps corresponding to inessential nodes in $T$ from $G^{(5)}$ preserves the (non-)planarity of $G^{(5)}$. Then we classify the potentially essential nodes as either essential or inessential. In Section 9.6.2, we define constraint graphs for the maximal chains of bicomps corresponding to inessential nodes, and show that replacing these chains by their constraint graphs preserves the (non-)planarity of the graph. In Section 9.6.3, we show that the resulting constraint graph $C_j$ of $G'_j$ has size $\mathcal{O}(|S_j|)$.

### 9.6.1 Discarding Inessential Bicomps

Let $C'_j$ be the graph obtained from $G'_j$ by replacing every bicomp $\mathcal{B}$ of $G'_j$ with its constraint graph $C_{\mathcal{B}}$. Let $T'$ be the tree obtained by removing all inessential nodes from $T$, and let $C''_j$ be the subgraph of $C'_j$ obtained by removing all constraint graphs $C_{\mathcal{B}_i}$ which correspond to bicomp nodes $\beta_i$ that were removed from $T$.

We define a sequence of graphs $G_0^{(6)}, \ldots, G_l^{(6)}$, where $G_0^{(6)} = G^{(5)}$ and $G_i^{(6)} = G_{i-1}^{(6)}[C'_i/C''_i]$, for $1 \leq i \leq l$. Graph $G^{(6)}$ is defined as $G^{(6)} = G_l^{(6)}$.

**Lemma 9.12** *Graph $G_i^{(6)}$ is planar if and only if graph $G_{i-1}^{(6)}$ is planar, for $1 \leq j \leq l$. A planar embedding of $G_{i-1}^{(6)}$ can be obtained from a planar embedding $\hat{G}_i^{(6)}$ of $G_i^{(6)}$ by locally replacing the embedding $\hat{C}''_j$ of $C''_j$ induced by $\hat{G}_i^{(6)}$ with a consistent planar embedding of $C'_j$.*

*Proof.* Graph $G_i^{(6)}$ is obtained from $G_{i-1}^{(6)}$ by removing vertices and edges from $G_{i-1}^{(6)}$. Thus, if $G_{i-1}^{(6)}$ is planar, $G_i^{(6)}$ is planar.

To show that $G_{i-1}^{(6)}$ is planar if $G_i^{(6)}$ is planar, we partition the graph $C'_j - C''_j$ into its connected components. Each such component $K$ is composed of inessential bicomps of $C'_j$ and shares only one cutpoint $v$ with $C''_j$. Hence, graph $K$ shares only vertex $v$ with $\bar{K}$ and can be embedded inside any face of $\hat{G}_i^{(6)}$ which has vertex $v$ on

its boundary. As this is true for all components of $C'_j - C''_j$, graph $G^{(6)}_{i-1}$ is planar if graph $G^{(6)}_i$ is planar. □

**Corollary 9.5** *Graph $G^{(6)}$ is planar if and only if graph $G^{(5)}$ is planar. A planar embedding of $G^{(5)}$ can be obtained from a planar embedding $\hat{G}^{(6)}$ of $G^{(6)}$ by replacing the embeddings of graphs $C''_1, \ldots, C''_l$ in $\hat{G}^{(6)}$ with consistent embeddings of graphs $C'_1, \ldots, C'_l$.*

Having disposed of the first set of bicomps corresponding to inessential nodes in $T$, we now classify the potentially essential nodes of $T'$ as either essential or inessential. A potentially essential node is essential if it has degree at least three in $T'$. Otherwise, it is inessential. Note that all inessential nodes have degree two, since all leaves of $T'$ and all internal nodes of degree at least three are essential. Thus, the set of inessential nodes in $T'$ can be partitioned into maximal paths. For each such path $P$ containing at least one bicomp node $\beta_i$, let $H_P = \mathcal{B}_{i_1} \cup \cdots \cup \mathcal{B}_{i_q}$, where $\beta_{i_1}, \ldots, \beta_{i_q}$ are the bicomp nodes in $P$. Next we replace each such graph $H_P$ with a constraint graph $C_P$ of constant size.

### 9.6.2 Compressing Chains of Inessential Bicomps

Let $H_P$ be a graph corresponding to a path $P$ of inessential nodes in $T'$. As all bicomps in $H_P$ are inessential, and every node in $P$ has degree two, graph $H_P$ shares exactly two vertices $a$ and $b$ with $\bar{H}_P$. If $H_P$ has an embedding such that vertices $a$ and $b$ are on the boundary of the same face, graph $C_P$ consists of the single edge $\{a, b\}$. Otherwise, $H_P$ is replaced by the graph $C_P$ shown in Figure 9.9. Graph $C_P$ is triconnected and has the property that vertices $a$ and $b$ are not on the boundary of the same face in the unique embedding $\hat{C}_P$ of $C_P$. The test which of the two cases applies can be carried out in linear time: If the graph $(V(H_P), E(H_P) \cup \{\{a, b\}\})$ is planar, there exists a planar embedding of $H_P$ such that vertices $a$ and $b$ appear on the same face. Otherwise, no such embedding exists.

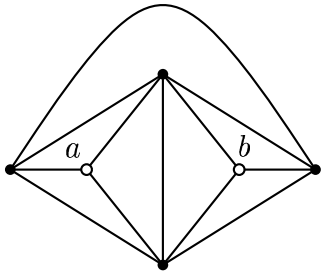



**Figure 9.9**
The constraint graph of a "twisted" chain of bicomps.

Let $T'_1, \ldots, T'_l$ be the trees obtained from trees $T_{\text{bic}}(G'_1), \ldots, T_{\text{bic}}(G'_l)$ using the construction in Section 9.6.1. Let $P_1, \ldots, P_q$ be the maximal paths of inessential vertices in trees $T'_1, \ldots, T'_l$, and let $H_{P_1}, \ldots, H_{P_q}$ be the subgraphs of $G^{(6)}$ induced by these paths. We define a sequence of graphs $G_0^{(7)}, \ldots, G_q^{(7)}$ as follows: $G_0^{(7)} = G^{(6)}$. For $1 \leq i \leq q$, $G_i^{(7)} = G_{i-1}^{(7)}[H_{P_i}/C_{P_i}]$. The approximate graph $A$ of $G$ is defined as $A = G_q^{(7)}$.

**Lemma 9.13** *Graph $G_i^{(7)}$ is planar if and only if graph $G_{i-1}^{(7)}$ is planar, for $1 \leq i \leq q$. A planar embedding of $G_{i-1}^{(7)}$ can be obtained from a planar embedding of $G_i^{(7)}$ by locally replacing the embedding $\hat{C}_{P_i}$ of graph $C_{P_i}$ with a consistent embedding of graph $H_{P_i}$.*

*Proof.* First assume that $G_{i-1}^{(7)}$ is planar. Let $\hat{G}_{i-1}^{(7)}$ be a planar embedding of $G_{i-1}^{(7)}$, and let $\hat{H}_{P_i}$ be the planar embedding of $H_{P_i}$ induced by $\hat{G}_{i-1}^{(7)}$. We partition $\bar{H}_{P_i}$ into maximal subgraphs such that each of these subgraphs is embedded in a different face of $\hat{H}_{P_i}$. As all bicomps in $H_{P_i}$ are inessential, such a subgraph can contain only cutpoints $a_i$ and $b_i$. If there are subgraphs of $\bar{H}_{P_i}$ containing both $a_i$ and $b_i$, $\hat{H}_{P_i}$ has a face with both vertices $a_i$ and $b_i$ on its boundary. Hence, $C_{P_i}$ consists of edge $\{a_i, b_i\}$, and all subgraphs of $\bar{H}_{P_i}$ containing $a_i$ and $b_i$ can be embedded without intersections in the only face of $\hat{C}_{P_i}$. The subgraphs of $\bar{H}_{P_i}$ containing only one of vertices $a_i$ and $b_i$ can be embedded inside any face of $\hat{C}_{P_i}$ which has the respective vertex on its boundary.

If $C_{P_i}$ is the graph shown in Figure 9.9, then $H_{P_i}$ does not have a planar embedding with both vertices $a_i$ and $b_i$ on the boundary of the same face. Hence, all subgraphs of $\bar{H}_{P_i}$ share at most one vertex $v$ with $H_{P_i}$. Each such subgraph can be embedded inside any face of $C_{P_i}$ which has vertex $v$ on its boundary. Hence, in both cases, $G_i^{(7)}$ is planar if $G_{i-1}^{(7)}$ is planar.

Now assume that $G_i^{(7)}$ is planar. Let $\hat{G}_i^{(7)}$ be a planar embedding of $G_i^{(7)}$. If $C_{P_i}$ consists of a single edge, then $H_{P_i}$ has an embedding $\hat{H}_{P_i}$ with vertices $a_i$ and $b_i$ on the same face. A simple transformation guarantees that the outer face of $\hat{H}_{P_i}$ has vertices $a_i$ and $b_i$ on its boundary. Then we replace edge $\{a_i, b_i\}$ in $\hat{G}_i^{(7)}$ with embedding $\hat{H}_{P_i}$. This produces a planar embedding of $G_{i-1}^{(7)}$, as $a_i$ and $b_i$ are the only vertices shared by $H_{P_i}$ and $\bar{H}_{P_i}$.

If $C_{P_i}$ is the graph shown in Figure 9.9, every component of $\bar{H}_{P_i}$ embedded inside a face of $\hat{C}_{P_i}$ contains only one of $a_i$ and $b_i$. Hence, it can be embedded inside any face of an embedding $\hat{H}_{P_i}$ of $H_{P_i}$ which has the respective vertex on its boundary. Thus, graph $G_{i-1}^{(7)}$ is planar if graph $G_i^{(7)}$ is planar. □

**Corollary 9.6** *Graph $A$ is planar if and only if graph $G^{(6)}$ is planar. A planar embedding of $G^{(6)}$ can be obtained from a planar embedding $\hat{A}$ of $A$ by locally replacing the embeddings $\hat{C}_{P_1}, \ldots, \hat{C}_{P_q}$ of graphs $C_{P_1}, \ldots, C_{P_q}$ with consistent embeddings of graphs $H_{P_1}, \ldots, H_{P_q}$.*

### 9.6.3 The Constraint Graph of the Component

The construction of the previous sections replaces each graph $G'_j$ with its constraint graph $C_j$. In this section, we show that $C_j$ is small.

**Lemma 9.14** *The constraint graph $C_j$ of graph $G'_j$ is a simple planar graph with $\mathcal{O}(|S_j|)$ vertices, for $1 \leq j \leq l$.*

*Proof.* The planarity of $C_j$ follows immediately from the above construction. In order to show that $C_j$ has $\mathcal{O}(|S_j|)$ vertices, let $T = T_{\text{bic}}(G'_j)$ be the bicomp-cutpoint-tree of $G'_j$. Let $T'$ be the tree constructed from $T$ in Section 9.6.1. We partition the

essential nodes of $T'$ into two classes: Type-I nodes are bicomp nodes $\beta_i$ such that $\mathcal{B}_i = \mathcal{B}(v)$, for some vertex $v \in S_j$. Type-II nodes have degree at least three in $T'$. There are at most $|S_j|$ type-I nodes. As in the proof of Lemma 9.11, the number of type-II nodes can be bounded by the number of leaves in $T'$, which is at most $|S_j|$ because all leaves of $T'$ are of type I. Thus, there are less than $2|S_j|$ essential nodes in $T'$.

Let $T''$ be the tree obtained from $T'$ by replacing every maximal path whose internal nodes are inessential by a single edge. Then the nodes of $T''$ are the essential nodes in $T'$, so that $T''$ has less than $2|S_j|$ nodes and less than $2|S_j| - 1$ edges. Every edge in $T''$ corresponds to a path $P$ in $T'$, which in turn corresponds to a possibly empty graph $H_P$ in $C'_j$. Each such graph $H_P$ is replaced by a constraint graph $C_P$ of constant size. Every essential bicomp in $C'_j$ contains at most as many cutpoints as edges incident to its corresponding node in $T''$. Thus, the total number of required vertices in all essential bicomps is less than $5|S_j|$. By Lemma 9.11, each such bicomp is replaced by a planar constraint graph whose size is linear in the number of its required vertices. Thus, $C_j$ has $\mathcal{O}(|S_j|)$ vertices. □

## 9.7 The Approximate Graph

The approximate graph $A$ of $G$ is the graph obtained after applying the replacement procedures of Sections 9.4 through 9.6 to $G$. The following two lemmas show that graph $A$ has the desired properties.

**Lemma 9.15** *Graph $A$ is planar if and only if graph $G$ is planar.*

*Proof.* This follows from Corollaries 9.1, 9.2, 9.3, and 9.4, Lemma 9.10, and Corollaries 9.5 and 9.6. □

**Lemma 9.16** *Graph $A$ has size $\mathcal{O}(N/(DB))$.*

*Proof.* In Step 4 of Algorithm 9.1, graph $G$ is partitioned into $\mathcal{O}\left(N/(DB)^2\right)$ graphs $G_1, \ldots, G_k$ such that $|\partial G_i| \leq DB$, for $1 \leq i \leq k$. In particular, the total boundary

size of all graphs $G_1, \ldots, G_k$ is $\sum_{i=1}^{k} |\partial G_i| = \mathcal{O}(N/(DB))$. Since graphs $G'_1, \ldots, G'_l$ are the connected components of graphs $\tilde{G}_1, \ldots, \tilde{G}_k$, this implies that $\sum_{j=1}^{l} |S_j| = \sum_{i=1}^{k} |\partial G_i| = \mathcal{O}(N/(DB))$. By Lemma 9.14, $|C_j| = \mathcal{O}(|S_j|)$, for $1 \leq j \leq l$, so that $\sum_{j=1}^{l} |C_j| = \mathcal{O}\left(\sum_{j=1}^{l} |S_j|\right) = \mathcal{O}(N/(DB))$. Graph $A$ consists of graphs $C_1, \ldots, C_l$ and the subgraph $G[S]$ of $G$ induced by the separator vertices in $S$. Graph $G[S]$ has $\mathcal{O}(N/(DB))$ vertices. Thus, $|A| \leq \sum_{j=1}^{l} |C_j| + |G[S]| = \mathcal{O}(N/(DB))$. □

## 9.8 Constructing the Final Embedding

Given a planar embedding of $A$, Lemma 9.15 implies that $G$ is planar. In this section, we discuss how to derive a planar embedding $\hat{G}$ of $G$ from a planar embedding $\hat{A}$ of the approximate graph $A$. Conceptually, the construction is fairly simple: We replace the embeddings of graphs $C_1, \ldots, C_l$ induced by $\hat{A}$ one by one with consistent embeddings of graphs $G'_1, \ldots, G'_l$. This intuitively simple procedure presents a few technicalities that have to be dealt with in order to obtain a valid embedding of $G$.

In Section 9.8.2, we define formally what we mean by an embedding of $G'_j$ which is "consistent" with an embedding of $C_j$ and show how to derive such an embedding. The "replacement" of embedding $\hat{C}_j$ with the computed embedding $\hat{G}'_j$ is done as follows: First we partition $\bar{C}_j$ into maximal subgraphs such that each of them is embedded inside a different face of $\hat{C}_j$. Then we place each such subgraph inside an appropriate face of $\hat{G}'_j$, without changing its embedding.

An important constraint on the replacement of graphs $C_1, \ldots, C_l$ with graphs $G'_1, \ldots, G'_l$ is given by the fact that it would take too many I/Os to extract the embedding of graph $C_j$ immediately before the replacement of $C_j$ with $G'_j$ takes place. Thus, we extract *all* embeddings $\hat{C}_1, \ldots, \hat{C}_l$ of graphs $C_1, \ldots, C_l$ induced by $\hat{A}$ before starting to replace them with graphs $G'_1, \ldots, G'_l$. However, the construction of embedding $\hat{G}'_j$ depends on the embedding $\hat{C}_j$ of $C_j$ at the time when $C_j$ is replaced with $G'_j$. Thus, we have to ensure that replacing embeddings $\hat{C}_1, \ldots, \hat{C}_{j-1}$ with embeddings $\hat{G}'_1, \ldots, \hat{G}'_{j-1}$ in $\hat{A}$ does not change the embedding $\hat{C}_j$ of $C_j$ induced by $\hat{A}$. The construction in Section 9.8.2 takes this into account.

Before we describe the construction of $\hat{G}$ from $\hat{A}$ in detail, we explain how the planar embedding $\hat{A}$ of $A$ and the final embedding $\hat{G}$ of $G$ are to be represented, and how to extract graphs $C_1, \ldots, C_l$ and their embeddings $\hat{C}_1, \ldots, \hat{C}_l$ I/O-efficiently.

### 9.8.1 Extracting the Embeddings of Constraint Graphs

Given the partition of $G$ into graphs $G'_1, \ldots, G'_j$ and $G[S]$, every edge in $G$ belongs to exactly one of these graphs. The separator algorithm of Chapter 8 can be augmented so that it labels every edge in $G$ with the name of the subgraph containing it. Similarly, every edge in $A$ belongs to one of the graphs $C_1, \ldots, C_l$ and $G[S]$. Edges in $G[S]$ are in both $G$ and $A$. The edges in $C_1, \ldots, C_l$ can easily be labelled as belonging to one of these graphs while constructing graphs $C_1, \ldots, C_l$ from graphs $G'_1, \ldots, G'_l$.

We represent embedding $\hat{A}$ of $A$ as a collection of *interlaced edge cycles*. That is, every edge $e = \{v, w\} \in A$ stores four pointers: two pointers $\text{succ}^v_A(e)$ and $\text{pred}^v_A(e)$ to its two neighbors clockwise and counterclockwise around $v$, respectively, and two pointers $\text{succ}^w_A(e)$ and $\text{pred}^w_A(e)$ to its two neighbors around $w$. It is easy to verify that a representation of a planar embedding using interlaced edge cycles can be transformed into any other standard representation of the planar embedding in $\mathcal{O}(\text{sort}(N))$ I/Os.

Our goal is to modify the pointers representing the embedding of $\hat{A}$ to obtain interlaced edge cycles representing a planar embedding $\hat{G}$ of $G$. In order to be able to extract the embedding of only a subset of the edges incident to any vertex in $A$, it is convenient to have the edges incident to each vertex $v$ numbered clockwise around $v$. That is, in addition to pointers $\text{pred}^v_A(e)$, $\text{succ}^v_A(e)$, $\text{pred}^w_A(e)$, and $\text{succ}^w_A(e)$, every edge $e = \{v, w\}$ in $A$ should store two labels $\nu_v(e)$ and $\nu_w(e)$ representing the numbers of edge $e$ in the clockwise orders of the edges around vertices $v$ and $w$, respectively. Such a labelling of the edges can be derived from the interlaced edge cycles representing $\hat{A}$ as follows:

We represent each edge $e = \{v, w\}$ in graph $A$ by two triples $(v, e, \text{succ}^v_A(e))$ and $(w, e, \text{succ}^w_A(e))$ and sort the resulting set of triples by their first components. The

result is a concatenation of lists, each representing a circular linked list of edges clockwise around a vertex of $A$. By replacing one of the triples $(v, e, \mathrm{succ}_A^v(e))$ in each list by the triple $(v, e, \mathbf{null})$, we obtain a collection of regular linked lists. Now we apply the list-ranking procedure of [43] to all of these lists simultaneously. The result is an assignment of a label $\nu_v(e)$ to each triple $(v, e, \cdot)$, where $\nu_v(e)$ is the number of edge $e$ in the order of edges clockwise around vertex $v$. Now it is sufficient to sort these triples by their second components and scan the resulting list to compute triples $(e, \nu_v(e), \nu_w(e))$, for all edges $e = \{v, w\}$ in $A$. This procedure takes $\mathcal{O}(\mathrm{sort}(N/(DB)))$ I/Os.

Graphs $C_1, \ldots, C_l$ and their embeddings $\hat{C}_1, \ldots, \hat{C}_l$ are now easily extracted: In order to extract graphs $C_1, \ldots, C_l$, we sort the edges in $A$ by their component labels. This produces a partition of $E(A)$ into sets $E(G[S]), E(C_1), \ldots, E(C_l)$. Given graph $C_j$, a representation of its embedding as interlaced edge cycles can be extracted by reversing the construction of the previous paragraph. In particular, we create two triples $(v, \nu_v(e), e)$ and $(w, \nu_w(e), e)$, for each edge $e = \{v, w\}$ in $C_j$, and sort the resulting list lexicographically. As a result, all edges incident to a vertex $v \in C_j$ are stored consecutively, sorted clockwise around $v$. In a single scan, we replace every triple $(v, \nu_v(e), e)$ with a triple $(e, \mathrm{pred}_{C_j}^v(e), \mathrm{succ}_{C_j}^v(e))$. Now we sort these triples by their first components and scan the resulting list to compute a list of quintuples $(e, \mathrm{pred}_{C_j}^v(e), \mathrm{succ}_{C_j}^v(e), \mathrm{pred}_{C_j}^w(e), \mathrm{succ}_{C_j}^w(e))$, for each edge $e = \{v, w\}$ in $C_j$. The whole construction takes $\mathcal{O}(\mathrm{sort}(N/(DB)))$ I/Os, for all graphs $C_1, \ldots, C_l$.

In order to facilitate the replacement of embedding $\hat{C}_j$ with an embedding $\hat{G}'_j$ of $G'_j$, we augment graph $C_j$ with edges $\mathrm{pred}_A^v(e)$, $\mathrm{succ}_A^v(e)$, $\mathrm{pred}_A^w(e)$, and $\mathrm{succ}_A^w(e)$, for each edge $e = \{v, w\} \in C_j$. Let $D_j$ be the graph obtained by augmenting $C_j$ in this manner, and let $\hat{D}_j$ be its planar embedding induced by $\hat{A}$. Observe that for every (copy of a) required vertex $v$ on the boundary of a face $f$ of $\hat{C}_j$, graph $D_j$ contains the first and last edges, $e'$ and $e''$, in clockwise order around $v$ which are embedded inside $f$. If $K$ is the subgraph of $\bar{C}_j$ embedded inside $f$, all edges in $K$ that are incident to $v$ appear between two such edges $e'$ and $e''$ in the clockwise

order around $v$. As embedding $\hat{C}_j$ is replaced with an embedding $\hat{G}'_j$ of $G'_j$ without changing the embedding of $K$, the edges in $E(D_j) \setminus E(C_j)$ are the only edges in $K$ whose neighbors in the embedding change as a consequence of the replacement, and all edges in $G'_j$ have their neighbors either in $G'_j$ or in $E(D_j) \setminus E(C_j)$. Thus, graph $D_j$ and its embedding $\hat{D}_j$ provide sufficient information to perform the replacement of $\hat{C}_j$ with $\hat{G}'_j$. Moreover, as every edge in $C_j$ has at most four neighbors in $\hat{A}$, $D_j$ contains at most five times as many edges as $C_j$, so that $D_j$ fits into internal memory, and the total size of graphs $D_1, \ldots, D_l$ is $\mathcal{O}\left(\sum_{j=1}^{l} |C_j|\right) = \mathcal{O}(N/(DB))$.

### 9.8.2 Replacing the Embedding of a Constraint Graph

Given graphs $D_1, \ldots, D_l$ and their embeddings $\hat{D}_1, \ldots, \hat{D}_l$, they can be used to replace the embeddings $\hat{C}_1, \ldots, \hat{C}_l$ of graphs $C_1, \ldots, C_l$ with consistent embeddings of graphs $G'_1, \ldots, G'_l$. This replacement is performed one graph at a time. In this section, we are concerned with deriving the embedding $\hat{G}'_j$ of $G'_j$ from $\hat{C}_j$ and replacing $\hat{C}_j$ with $\hat{G}'_j$. In Section 9.8.5, we show how to exchange the necessary information about updates of the interlaced edge cycles resulting from these replacements between graphs $D_1, \ldots, D_l$ so that subsequent replacements can be performed correctly.

Given an embedding $\hat{C}_j$ of $C_j$, the goal of the construction in this section is to construct an embedding $\hat{G}'_j$ of $G'_j$ so that the subgraphs of $\bar{C}_j$ embedded in the faces of $\hat{C}_j$ can be embedded inside appropriate faces of $\hat{G}'_j$. This goal is achieved by undoing the compression steps of Sections 9.4 through 9.6, one by one, and maintaining a planar embedding of the current graph as well as a mapping of the subgraphs of $\bar{C}_j$ to the faces of the embedding. We call the current embedding and the mapping of subgraphs of $\bar{C}_j$ to the faces of the embedding *consistent* with $\hat{C}_j$ if the following invariant holds:

(I1) Let $K_1, \ldots, K_s$ be the maximal subgraphs of $\bar{C}_j$ so that each of them is embedded inside a different face of $\hat{C}_j$. Then each of the graphs $K_1, \ldots, K_s$ is embedded completely inside one face of the current embedding. Let $e_1, \ldots, e_q$ be the edges in $E(K_1) \cup \cdots \cup E(K_s)$ incident to a vertex $v \in C_j$. If edges

$e_1, \ldots, e_q$ appear in this order clockwise around $v$ in $\hat{A}$, then edges $e_1, \ldots, e_q$ appear in the same order clockwise around $v$ in the current embedding.

This invariant ensures that the embeddings of graphs $C_{j+1}, \ldots, C_l$ are not changed by the replacement of $C_j$ with $G'_j$. Given that the embeddings of graphs $K_1, \ldots, K_s$ are not modified when replacing $\hat{C}_j$ with $\hat{G}'_j$, the following invariant is equivalent to Invariant (I1):

(I2) Let $E_1, \ldots, E_s$ be the maximal subsets of $E(D_j) \setminus E(C_j)$ such that the edges in each subset are embedded inside a different face of $\hat{C}_j$. Then the edges in each of these subsets are embedded inside the same face of the current embedding. Let $e_1, \ldots, e_q$ be the edges in $E(D_j) \setminus E(C_j)$ incident to a vertex $v \in C_j$. If edges $e_1, \ldots, e_q$ appear in this order clockwise around $v$ in $\hat{D}_j$, then edges $e_1, \ldots, e_q$ appear in the same order clockwise around $v$ in the current embedding.

In the next section, we provide the details of the reconstruction of $G'_j$ from $C_j$ along with an embedding $\hat{G}'_j$ of $G'_j$. In Section 9.8.4, we discuss the changes to the interlaced edge cycles representing $\hat{A}$ that need to be made in order to obtain interlaced edge cycles representing the embedding $\hat{A}[C_j/G'_j]$ of graph $A[C_j/G'_j]$ which is obtained when replacing $\hat{C}_j$ with the constructed embedding $\hat{G}'_j$.

### 9.8.3 Constructing Local Embeddings

In this section, a planar embedding $\hat{G}'_j$ of graph $G'_j$ is obtained by undoing the compression steps used to construct the constraint graph $C_j$ from $G'_j$ and at all times maintaining a planar embedding of the current graph.

**Introducing parallel edges.** Recall that the construction of the constraint graphs of bicomps removes all parallel edges that may have been introduced by the removal of inessential tricomps. These edges need to be re-introduced, so that they can later be replaced by the (group of) tricomps they represent. For every group of parallel edges with endpoints $v$ and $w$, the required additional copies of edge $\{v, w\}$ are embedded

parallel to the only edge $\{v, w\}$ in $C_j$ so that the (degenerate) faces bounded by these edges do not contain any vertices. As this changes neither the order of edges in $E(D_j) \setminus E(C_j)$ around their endpoints nor places edges that were in the same face into different faces, Invariant (I2) is preserved.

**Replacing the embedding of a triconnected component.** The next step is to replace the embeddings $\hat{C}^\circ_{\mathcal{T}_1}, \ldots, \hat{C}^\circ_{\mathcal{T}_q}$ of the kernels of the constraint graphs $C_{\mathcal{T}_1}, \ldots, C_{\mathcal{T}_q}$ of essential or separating tricomps $\mathcal{T}_1, \ldots, \mathcal{T}_q$ in $G'_j$ with embeddings $\hat{\mathcal{T}}^\circ_1, \ldots, \hat{\mathcal{T}}^\circ_q$ of the kernels of these tricomps.

The treatment of tricomp $\mathcal{T}$ depends on its type. If $\mathcal{T}$ is a bond, then $C_\mathcal{T} = \mathcal{T}$, so that nothing needs to be done. If $\mathcal{T}$ is a cycle, every path in $\mathcal{T}$ whose internal vertices are not required is represented by a single edge in $C_\mathcal{T}$. Now we reverse this operation, replacing each such edge by its corresponding path. This replacement can easily be done in a manner that preserves Invariant (I2). The process of replacing the embedding $\hat{C}^\circ_\mathcal{T}$ of $C^\circ_\mathcal{T}$ with an embedding $\hat{\mathcal{T}}^\circ$ of $\mathcal{T}^\circ$ is only slightly more complicated in the case when $\mathcal{T}$ is a triconnected simple graph.

First recall that in this case, $\mathcal{T}$ and $C_\mathcal{T}$ both have a unique planar embedding. The construction of $C_\mathcal{T}$ from $\mathcal{T}$ in Section 9.4 preserves the order of faces with at least two required vertices on their boundaries around the required vertices of $\mathcal{T}^\circ$. Hence, every subgraph of $\bar{C}^\circ_\mathcal{T}$ embedded inside such a face of $\hat{C}^\circ_\mathcal{T}$ can be embedded inside the corresponding face of $\hat{\mathcal{T}}^\circ$. This preserves the order of edges in these graphs incident to a required vertex of $\mathcal{T}^\circ$ clockwise around that vertex. Any graph embedded inside a face of $\hat{C}^\circ_\mathcal{T}$ with only one required vertex $v$ on its boundary shares only vertex $v$ with $C^\circ_\mathcal{T}$. Hence, it can be embedded inside any face of $\hat{\mathcal{T}}^\circ$ which has vertex $v$ on its boundary. This gives us enough freedom to embed these subgraphs in a way that preserves the order of all edges in $\bar{C}^\circ_\mathcal{T}$ around the required vertices in $C^\circ_\mathcal{T}$. Thus, Invariant (I2) is preserved.

**Replacing the embedding of a core.** The next step is the replacement of the embeddings of the constraint graphs of cores with embeddings of the cores. Let $\mathcal{C}$

be a core. We treat the three different cases depicted in Figure 9.8 separately. In Case (a), two edges in the current embedding need to be replaced by the embeddings of two graphs, each sharing two vertices with the rest of the graph. This is easily done in a manner preserving Invariant (I2). In Case (b), the whole graph $\bar{C}_{\mathcal{C}}$ is embedded in the outer face of $C^{\circ}_{\mathcal{C}}$. Thus, we choose an embedding of $\mathcal{C}$ such that virtual edges $(a, b, i)$ and $(c, d, j)$ are on its outer boundary and then embed $\bar{C}_{\mathcal{C}}$ in the outer face of $\hat{\mathcal{C}}^{\circ}$. In Case (c), finally, graph $\bar{C}_{\mathcal{C}}$ consists of two subgraphs, one of which is embedded in the interior face of $\hat{C}^{\circ}_{\mathcal{C}}$; the other is embedded in the exterior face of $C^{\circ}_{\mathcal{C}}$. Since these two subgraphs do not share any vertices, embedding each of them in the corresponding face of $\hat{\mathcal{C}}$ maintains Invariant (I2). Thus, in all three cases, $\hat{C}_{\mathcal{C}}$ can be replaced with an embedding of $\mathcal{C}$ in a manner that preserves Invariant (I2).

**Replacing the embedding of a fan.** Next the embeddings of the constraint graphs of fans are replaced with embeddings of the respective fans. As we did for cores, we distinguish the different possible configurations shown in Figure 9.7. In Cases (a)–(c), edges in the constraint graphs on the right have to be replaced with the corresponding subgraphs on the left. Each of these subgraphs shares only the two endpoints of the corresponding edge with the rest of $D_j$. Thus, this replacement can easily be done in a way that preserves Invariant (I2). In Cases (d)–(f), the faces with at least two required vertices on their boundaries appear in the same order around these required vertices in the embedding of the fan and the embedding of its constraint graph. Thus, embedding all graphs embedded in faces of $\hat{C}_{\mathcal{F}}$ with at least two required vertices on their boundaries inside the corresponding faces of $\hat{\mathcal{F}}$ preserves Invariant (I2). As in the case of a triconnected component, a graph embedded inside a face of $\hat{C}_{\mathcal{F}}$ with at most one required vertex on its boundary can be embedded inside any face of $\hat{\mathcal{F}}$ with that vertex on its boundary. Hence, these graphs can be arranged so that Invariant (I2) is not violated.

**Introducing inessential tricomps.** Inessential tricomps that have been removed from $G$ in Section 9.5.1 have been grouped into maximal groups corresponding to

complete subtrees in the tricomp tree of the bicomp containing them. The subgraph $K$ obtained by merging the tricomps in such a group shares exactly one virtual edge $(a, b, i)$ with the rest of $G$, and has consequently been replaced by a non-virtual edge $(a, b, i)$. In order to re-introduce this subgraph into the embedding, edge $(a, b, i)$ has to be replaced with an embedding of $K^\circ$ that has vertices $a$ and $b$ on the outer face. Such an embedding exists, by the planarity of $K$. Replacing edge $(a, b, i)$ with such an embedding preserves Invariant (I2).

**Replacing the embedding of a chain of inessential bicomps.** Having dealt with the embeddings of essential and inessential tricomps, all inessential bicomps that have been removed from $G$ have to be re-introduced. The first group of inessential bicomps are those that have been replaced by constraint graphs of constant size in Section 9.6.2. The second group are those that have been completely removed from $G$ in Section 9.6.1. The bicomps in the first group form chains such that each chain $K$ shares exactly two vertices, $a$ and $b$, with the rest of $G$. Depending on whether $a$ and $b$ can appear on the same face of an embedding of $K$, $K$ has been replaced by a single edge $\{a, b\}$ or by a constraint graph of constant size which does not allow $a$ and $b$ to appear on the boundary of the same face. In the former case, edge $\{a, b\}$ has to be replaced with an embedding of $K$ with vertices $a$ and $b$ on the outer face. As before, this replacement preserves Invariant (I2). In the latter case, no subgraph of $\bar{K}$ can contain both $a$ and $b$. We choose any embedding $\hat{K}$ of $K$ and embed the subgraphs of $\bar{K}$ incident to vertices $a$ and $b$ in the faces of $\hat{K}$ incident to these vertices in a manner that preserves their order around these vertices.

**Introducing inessential bicomps.** Finally, all bicomps that have been removed completely from $G'_j$ have been grouped into maximal connected subgraphs such that each such subgraph shares only one vertex with the rest of $G$. Each such subgraph $K$ sharing a vertex $v$ with $\bar{K}$ can be embedded inside any face of the current embedding which has vertex $v$ on its boundary, without violating Invariant (I2).

### 9.8.4 Updating the Interlaced Edge Cycles

After finishing the replacement steps described above, we obtain an embedding $\hat{G}'_j$ of $G'_j$ and an embedding of the edges in $E(D_j) \setminus E(C_j)$ inside the faces of $\hat{G}'_j$. It remains to integrate this information with the embedding $\hat{A}$ of $A$, in order to obtain an embedding $\hat{A}[C_j/G'_j]$ of $A[C_j/G'_j]$. In $\hat{A}[C_j/G'_j]$, every edge $e$ in $G'_j$ has both its neighbors around both its endpoints in $E(G'_j) \cup (E(D_j) \setminus E(C_j))$. Thus, if $G'$ is the graph induced by the edges in $E(G'_j) \cup (E(D_j) \setminus E(C_j))$, and $\hat{G}'$ is the embedding of $G'$ derived by the above construction, then the pointers to be stored with $e$ in the interlaced edge cycles representing $\hat{A}[C_j/G'_j]$ are the same as the ones stored with $e$ in the interlaced edge cycles representing $\hat{G}'$. Similarly, as our construction ensures that the embeddings of subgraphs of $\bar{C}_j$ embedded inside different faces of $\hat{C}_j$ are not changed, all edges in $E(\bar{C}_j) \setminus E(D_j)$ have the same neighbors in $\hat{A}[C_j/G'_j]$ as in $\hat{A}$. Finally, let $e = \{v, w\}$ be an edge in $E(D_j) \setminus E(C_j)$. We discuss the necessary updates for edge $e$, using its pointer $\text{succ}^v_A(e)$ to its successor clockwise around $v$ as an example. The other three pointers are updated in a similar fashion.

Observe that since $e \in E(D_j) \setminus E(C_j)$, either $\text{succ}^v_A(e) \in C_j$ or $\text{pred}^v_A(e) \in C_j$, or both. Let $K$ be the subgraph of $\bar{C}_j$ containing edge $e$ and let $A' = A[C_j/G'_j]$. If $\text{succ}^v_A(e) \in C_j$, then $\text{succ}^v_{A'}(e) = \text{succ}^v_{G'}(e)$. Otherwise, $\text{succ}^v_A(e) \in K$. As the embedding of graph $K$ is not modified by the replacement of $\hat{C}_j$ with $\hat{G}'_j$, $\text{succ}^v_{A'}(e) = \text{succ}^v_A(e)$ in this case.

### 9.8.5 Iterative Replacement of Subgraphs

In the previous section, we have shown how to derive a planar embedding $\hat{A}[C_j/G'_j]$ of graph $A[C_j/G'_j]$ from an embedding $\hat{A}$ of graph $A$ by locally modifying the predecessor and successor pointers of edges in $D_j$ and $G'_j$. Now we use the procedure described in the previous section to produce a sequence of graphs $A_0, \ldots, A_l$ and embeddings $\hat{A}_0, \ldots, \hat{A}_l$ of these graphs, where $A_0 = A$ and $A_i = A_{i-1}[C_i/G'_i]$, for $1 \leq i \leq l$. Embedding $\hat{A}_0$ is the embedding $\hat{A}$ of graph $A_0 = A$. The embedding $\hat{A}_i$ of graph $A_i$ is computed from the embedding $\hat{A}_{i-1}$ of graph $A_{i-1}$ by applying the procedure of

the previous section. The final graph $A_l$ is graph $G$, so that $\hat{G} = \hat{A}_l$ is the desired embedding of graph $G$.

There are two issues that need to be addressed, in order to make this iterative replacement of graphs $C_1, \ldots, C_l$ with graphs $G'_1, \ldots, G'_l$ work: (1) When replacing $\hat{C}_i$ with $\hat{G}'_i$, the neighbors of the edges in $C_i$ clockwise and counterclockwise around their endpoints are not necessarily the same in $\hat{A}_{i-1}$ as in $\hat{A}$. Hence, graph $D_i$ needs to be updated, in order to correctly represent this neighborhood information, before it can be used to construct $\hat{G}'_i$ from $\hat{C}_i$. (2) Let $p_v, s_v, p_w, s_w$ be the four neighbors of an edge $e \in G'_i$ in the embedding $\hat{A}_i$ obtained after replacing $\hat{C}_i$ with $\hat{G}'_i$. If one of these neighbors, say $p_v$, is contained in a graph $C_j$, $j > i$, then $p_v$ will be replaced by another edge $p'_v$ when graph $C_j$ is replaced with graph $G'_j$. Thus, immediately after replacing $C_i$ and $G'_i$, the counterclockwise neighbor of edge $e$ around vertex $v$ in the final embedding $\hat{G}$ is not known. Hence, we need a criterion to decide when the neighborhood relationship between two edges cannot be broken as a result of subsequent replacements, so that pointers between these two edges can be added to the final embedding. We address these two problems next.

#### 9.8.5.1 Updating the Augmented Constraint Graph

In order to update graph $D_i$ so that it contains the correct neighbors of the edges in $C_i$ in the current embedding $\hat{A}_{i-1}$ of $A_{i-1}$, we use a priority queue $Q$ to collect information about the necessary updates of $D_i$. Before replacing $C_i$ with $G'_i$, we retrieve this information from $Q$ and update $D_i$ appropriately.

Recall that every edge in $A$ is labelled as belonging to one of the graphs $C_1, \ldots, C_l$ or $G[S]$. If there is an edge $e = \{v, w\} \in E(D_j) \setminus E(C_j)$ such that $e \in E(C_i)$, for $i > j$, and $\text{pred}^v(e)$ changes as a result of replacing $C_j$ with $G'_j$, we put a quintuple $(e, v, \text{“pred”}, j, \text{pred}^v(e))$ into $Q$ and give it priority $i$. For a successor change, the quintuple is of the form $(e, v, \text{“succ”}, j, \text{succ}^v(e))$. When replacing graph $C_i$ with graph $G'_i$, we retrieve all quintuples with priority $i$ from $Q$. As graphs $C_1, \ldots, C_{i-1}$ have already been replaced with graphs $G'_1, \ldots, G'_{i-1}$ when this happens, all entries

with lower priority have already been retrieved from $Q$. Hence, retrieving all entries with priority $i$ from $Q$ amounts to repeated application of DeleteMin operations until the first entry with priority greater than $i$ is retrieved. This entry is then put back into $Q$.

Next we use the quintuples retrieved from $Q$ to update graph $D_i$: We sort the list of retrieved quintuples lexicographically, so that for each edge $e = \{v, w\}$, all quintuples $(e, v, \text{“pred”}, \cdot, \cdot)$ and all quintuples $(e, v, \text{“succ”}, \cdot, \cdot)$ are stored consecutively. We sort these quintuples by their fourth component, which can be interpreted as the time when this quintuple was queued. Hence, the quintuple with the largest fourth component is the most recent update of the predecessor (or successor) of edge $e$ in the clockwise order around vertex $v$, so that its fifth component represents the correct predecessor (or successor) of $e$ around $v$ in $\hat{A}_{i-1}$. We scan the sorted list and discard all quintuples $(e, v, \text{“pred”}, \cdot, \cdot)$ and $(e, v, \text{“succ”}, \cdot, \cdot)$, except the last one, for each pair $(e, v)$. The result is a list $L_i$ containing quintuples $(e, v, \text{“pred”}, \cdot, \text{pred}^v_{A_{i-1}}(e))$ and $(e, v, \text{“succ”}, \cdot, \text{succ}^v_{A_{i-1}}(e))$, for all edges whose neighbors around their endpoints are different in $\hat{A}$ and $\hat{A}_{i-1}$. The size of this list is at most the size of $D_i$. Hence, graphs $G'_i$, $D_i$, and list $L_i$ fit into internal memory. We use list $L_i$ to update graph $D_i$, and then proceed to the construction of embedding $\hat{G}'_i$ from $\hat{C}_i$, as described in Section 9.8.2.

The total number of I/Os spent on queuing and dequeuing quintuples in $Q$, as well as sorting the retrieved entries before replacing graph $C_i$ with graph $G'_i$, is $\mathcal{O}(\text{sort}(T))$, for all graphs $G'_1, \ldots, G'_l$, where $T$ is the total number of quintuples produced by changing the neighbors of edges in sets $E(D_i) \setminus E(C_i)$, $1 \leq i \leq l$. This number, however, is proportional to the total number of edges in sets $E(D_i) \setminus E(C_i)$, $1 \leq i \leq l$, as the replacement of graph $C_i$ with graph $G'_i$ can change at most all four neighbors of an edge in $E(D_i) \setminus E(C_i)$. Hence, $T = \mathcal{O}(N/(DB))$, and it takes $\mathcal{O}(\text{sort}(N/(DB)))$ I/Os to maintain graphs $D_1, \ldots, D_l$ and their embeddings, which provide the required information to replace embeddings $\hat{C}_1, \ldots, \hat{C}_l$ with embeddings $\hat{G}'_1, \ldots, \hat{G}'_l$.

#### 9.8.5.2 Adding Pointers to the Final Edge Lists

It remains to address the problem of producing the interlaced edge cycles representing the final embedding $\hat{G}$ of graph $G$. The problem is that the successor of an edge $e \in G'_i$ is changed after replacing $C_i$ with graph $G'_i$ if this successor is an edge $e' \in C_j$, $j > i$. This change of successor happens when graph $C_j$ is replaced with graph $G'_j$. Thus, when graph $C_i$ is replaced with graph $G'_i$, the successor of edge $e$ cannot be written to disk yet. The following simple criterion guarantees that the neighbor pointers for every edge $e$ are written only when they do not change any more: If an edge $e \in G'_i$ has its successor in a graph $C_j$, $j > i$, this successor will change. Thus, the successor is not written to disk at this point. If on the other hand, edge $e$ has its successor in a graph $G'_j$, $j \leq i$, or in $G[S]$, then this neighborhood relation cannot change any more as a result of replacing graphs $C_{i+1}, \ldots, C_l$ with graphs $G'_{i+1}, \ldots, G_l$. This is true because edge $e$ and its successor are embedded inside the same faces of embeddings $\hat{C}_{i+1}, \ldots, \hat{C}_l$, and the embeddings of the maximal subgraphs embedded inside the faces of embedding $\hat{C}_j$ do not change when replacing $C_j$ with $G'_j$. Thus, the successor pointer of $e$ and the predecessor pointer of its successor can be written to disk as representing the neighborhood relationship in the final embedding $\hat{G}$ of $G$.

Once all constraint graphs $C_1, \ldots, C_l$ have been replaced with graphs $G'_1, \ldots, G'_l$, the algorithm has produced a list of $4|E|$ quadruples $(e, v, \text{“pred”}, \text{pred}^v_G(e))$ and $(e, v, \text{“succ”}, \text{succ}^v_G(e))$, four quadruples per edge. We sort these quadruples lexicographically, so that the quadruples representing the four neighbors of an edge $e$ are stored consecutively. Now a scan of this sorted list suffices to produce the interlaced edge cycles containing quintuples $(e, \text{pred}^v_G(e), \text{succ}^v_G(e), \text{pred}^w_G(e), \text{succ}^w_G(e))$ which represent the final embedding $\hat{G}$ of $G$. We summarize this section in the following lemma, which concludes the proof of Theorem 9.2.

**Lemma 9.17** *Given a planar embedding $\hat{A}$ of the approximate graph $A$, the local replacement of embeddings $\hat{C}_1, \ldots, \hat{C}_l$ with consistent embeddings $\hat{G}'_1, \ldots, \hat{G}'_l$ can be performed in $\mathcal{O}(\text{sort}(N))$ I/Os using linear space. The result is a planar embedding $\hat{G}$ of graph $G$.*

## 9.9 A Lower Bound for Planar Embedding

By Lemma 2.1, the I/O-complexity of our planar embedding algorithm can be reduced to $\mathcal{O}(\text{perm}(N))$. In this section, we prove a matching lower bound. The proof of the lower bound uses a reduction from the problem of permuting a list of $N$ items $x_1, \ldots, x_N$. In particular, we show that if a representation of a planar embedding of a planar graph $G$ as interlaced edge cycles can be computed in $o(\text{perm}(N))$ I/Os, then the desired permutation of items $x_1, \ldots, x_N$ can be computed in $o(\text{perm}(N))$ I/Os. Since it is shown in [172] that permuting $N$ data items takes $\Omega(\text{perm}(N))$ I/Os, this implies that planar embedding requires $\Omega(\text{perm}(N))$ I/Os.

**Lemma 9.18** *If there is an algorithm $\mathcal{A}$ that computes a planar embedding of a planar graph with $N$ vertices in $\mathcal{O}(\mathcal{I}(N))$ I/Os, then there exists an algorithm $\mathcal{A}'$ that permutes a list of $N$ data items in $\mathcal{O}(\mathcal{I}(N))$ I/Os.*

*Proof.* We assume that the input to algorithm $\mathcal{A}'$ is given as follows: Let $x_1, \ldots, x_N$ be $N$ data items to be arranged in the order $x_{\sigma(1)}, \ldots, x_{\sigma(N)}$, for some permutation $\sigma : [1, N] \to [1, N]$. Algorithm $\mathcal{A}'$ is provided with two lists $L_1 = \langle (1, x_1), \ldots, (N, x_N) \rangle$ and $L_2 = \langle (\sigma(1), y_1), \ldots, (\sigma(N), y_N) \rangle$. The goal of algorithm $\mathcal{A}'$ is to compute a list $L = \langle (x_{\sigma(1)}, y_1), \ldots, (x_{\sigma(N)}, y_N) \rangle$. In order to achieve this, algorithm $\mathcal{A}'$ computes a graph $G$ whose planar embedding is unique, and such that list $L$ can be extracted from the interlaced edge cycles representing the planar embedding $\hat{G}$ of $G$ in $\mathcal{O}(\mathcal{I}(N))$ I/Os. The construction of graph $G$ takes $\mathcal{O}(\text{scan}(N))$ I/Os, so that list $L$ can be computed in $\mathcal{O}(\mathcal{I}(N))$ I/Os by constructing $G$, computing interlaced edge cycles representing $\hat{G}$, and extracting $L$ from $\hat{G}$.

The vertex set $V$ of graph $G$ consists of four sets $V_0, V_1, V_2, V_3$: Set $V_0$ contains a special central vertex $z$. Set $V_1$ contains all elements of $L_1$. Set $V_2$ contains all elements of $L_2$. Set $V_3$ contains $2N$ vertices numbered 1 through $2N$. The edge set $E$ of $G$ contains edges $\{v, z\}$, for all $v \in V_1 \cup V_2$, edges $\{2i - 1, (i, x_i)\}$, for $1 \leq i \leq N$, edges $\{2\sigma(i), (\sigma(i), y_i)\}$, for $1 \leq i \leq N$, and edges $\{1, 2\}, \{2, 3\}, \ldots, \{2N - 1, 2N\}, \{2N, 1\}$. Graph $G$ is shown in Figure 9.10.

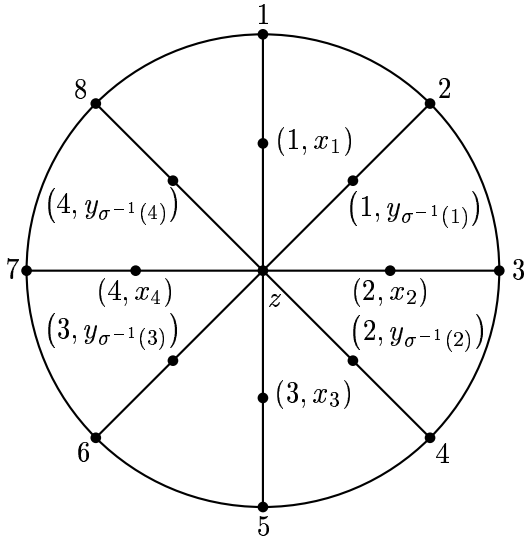



**Figure 9.10**
The graph $G$ constructed in the proof of Lemma 9.18.

The vertex set of $G$ can be constructed in $\mathcal{O}(\text{scan}(N))$ I/Os by appending vertex $z$ and vertices $1, \ldots, 2N$ to the concatenation of lists $L_1$ and $L_2$. In order to represent the edge set of $G$, algorithm $\mathcal{A}'$ computes the adjacency lists of all vertices in $G$. The adjacency list of every vertex $(i, x_i) \in V_1$ contains vertices $2i - 1$ and $z$. These lists can easily be constructed in a single scan over list $L_1$. The adjacency list of every vertex $(\sigma(i), y_i) \in V_2$ contains vertices $2\sigma(i)$ and $z$. It takes a single scan over list $L_2$ to construct these lists. The adjacency list of vertex $z$ is the concatenation of lists $L_1$ and $L_2$, which can be produced in $\mathcal{O}(\text{scan}(N))$ I/Os. The adjacency list of a vertex $2i - 1$, $1 \leq i \leq N$, contains vertices $2i - 2$, $2i$, and $(i, x_i)$. These lists can be produced in a single scan over list $L_1$. The adjacency list of a vertex $2\sigma(i)$, $1 \leq i \leq N$, contains vertices $2\sigma(i) - 1$, $2\sigma(i) + 1$, and $(\sigma(i), y_i)$. These lists can be produced in a single scan over list $L_2$. Thus, graph $G$ can be constructed in $\mathcal{O}(\text{scan}(N))$ I/Os. Moreover, the embedding of graph $G$ shown in Figure 9.10 is unique. Hence, after computing $\hat{G}$, $\text{succ}^z(\{z, (i, x_i)\}) = \{z, (i, y_{\sigma^{-1}(i)})\}$. Algorithm $\mathcal{A}'$ scans the interlaced edge cycles representing $\hat{G}$ and discards all quintuples except those belonging to edges $\{z, v\}$, $v \in V_1$. Each remaining quintuple is transformed

into the pair $(x_i, y_{\sigma^{-1}(i)})$. The resulting list is a permutation of the desired list $L$. In order to compute list $L$, algorithm $\mathcal{A}'$ reverses the I/O-operations that have been performed to arrange items $y_1, \ldots, y_N$ in the current order. This reversal performs the same number of I/O-operations as the construction of $\hat{G}$, i.e., $\mathcal{O}(\mathcal{I}(N))$ I/Os. Hence, list $L$ can be extracted from $\hat{G}$ in $\mathcal{O}(\text{scan}(N) + \mathcal{I}(N)) = \mathcal{O}(\mathcal{I}(N))$ I/Os. □

**Corollary 9.7** *It takes $\Omega(\text{perm}(N))$ I/Os to compute an embedding of a planar graph with $N$ vertices, if this embedding is to be represented as interlaced edge cycles.*

**Remark.** During the thesis defense, Roberto Tamassia brought previous work on dynamic planarity testing to our attention. In [68], a hierarchical separator decomposition is used to obtain a data structure that can be maintained under edge insertions and deletions in $\mathcal{O}\left(\sqrt{N}\right)$ amortized time and can be used to decide within the same time bound whether the current graph is planar. The construction of the data structure is based on compressed certificates for planarity (see [68, 82] for a definition of certificates for graph properties). The constraint graph $C_i$ we construct for each subgraph $G'_i$ in our algorithm is such a certificate. The construction of constraint graph $C_i$ presented here has previously been proposed in [82], where it is used to obtain a data structure for dynamic planarity testing which supports updates and queries in $\mathcal{O}(N^{2/3})$ time. The algorithm of [68] uses the certificates produced by the construction of [82] and improves the separator decomposition to obtain a better update and query bound.

# Chapter 10

# Applications of Planar Separators

In this chapter, we apply Theorems 8.3 and 9.2 and existing results of [9, 12] to obtain $\mathcal{O}(\text{sort}(N))$ I/O algorithms for the single source shortest path problem on planar graphs with non-negative edge weights, BFS in planar graphs, computing $\varepsilon$-vertex separators of low cost for planar graphs whose vertices have non-negative costs and non-negative weights, and computing $\varepsilon$-edge separators of weighted planar graphs.

## 10.1 Breadth-First Search and Single Source Shortest Paths

In this section, we discuss our solution to the single-source shortest path problem. Breadth-first search can be solved using the same algorithm after giving every edge in the graph the same weight. Our SSSP-algorithm is based on the following result.

**Theorem 10.1 (Arge et al. [12])** *Given a regular proper $(DB)^2$-partition of a directed planar graph $G = (V, E)$ with non-negative edge weights and bounded degree, the single-source shortest path problem on $G$ can be solved in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space.*

In order to be able to apply this result to solve SSSP on arbitrary planar graphs, we have to show how to satisfy the two conditions of the lemma: (1) We have to transform

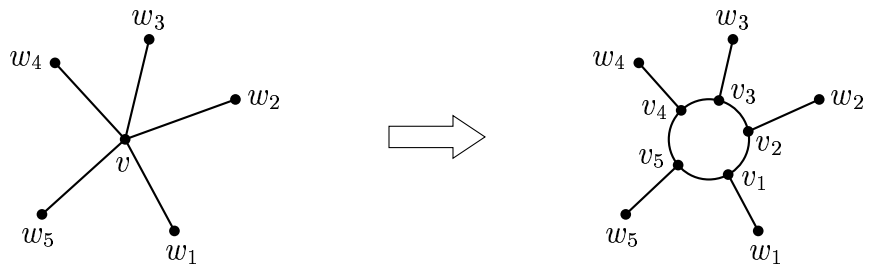



**Figure 10.1**
Transforming a vertex of high degree into a cycle of degree-3 vertices.

any planar graph into a planar graph of bounded degree so that the distances between vertices are essentially preserved. (2) We have to modify our separator algorithm of Chapter 8 so that the produced partition is regular.

**Constructing an equivalent planar graph of bounded degree.** To solve the first problem, we apply the following transformation: First we compute a planar embedding of $G$ using Algorithm 9.1. Given the computed embedding $\hat{G}$ of $G$, we replace every vertex $v$ of degree $d > 3$ in $G$ with a cycle $C_v = (v_1, \ldots, v_d)$ and connect every neighbor $w_i$, $1 \leq i \leq d$, of $v$ to a distinct vertex in the cycle so that the neighbors of vertices $w_1, \ldots, w_k$ appear in the same order along $C_v$ as edges $\{v, w_1\}, \ldots, \{v, w_k\}$ clockwise around $v$ (Figure 10.1). Denote the resulting graph by $G'$. If every edge in the created cycles is assigned weight zero, it is easy to show that the distance between any two vertices in $G$ is the same as the distance between any two vertices in the corresponding cycles in $G'$. The total number of vertices in $G'$ is bounded by twice the number of edges in $G$. Hence, $G'$ has $\mathcal{O}(N)$ vertices. Given a representation of $\hat{G}$ using interlaced edge cycles, the construction of graph $G'$ from graph $G$ can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os. The details are straightforward.

**Computing a regular partition.** Given a planar graph $G'$ of degree three, we modify Algorithm 8.1 as follows so that it computes a regular proper $h$-partition $\mathcal{P} = (S, \{G_1, \ldots, G_k\})$ of $G'$. Separator $S$ is computed as before. Only the grouping of connected components to produce subgraphs $G_1, \ldots, G_k$ is changed.

Let $Q_1, \ldots, Q_r$ be the connected components of $G' - S$. Then we construct a graph $H$ containing one vertex $v_i$ per connected component $Q_i$ and an edge between two vertices $v_i$ and $v_j$ if $\partial Q_i \cap \partial Q_j \neq \emptyset$. Since the vertices in $G'$ have degree at most three, graph $H$ is planar, and $\sum_{i=1}^{r} |\partial Q_i| = \mathcal{O}(|S|) = \mathcal{O}\left(N/\sqrt{h}\right)$. We assign weights $\omega(v_i) = |Q_i|$ and $\gamma(v_i) = |\partial Q_i|$ to every vertex $v_i$, $1 \leq i \leq r$, and contract edges in $H$ until no two adjacent vertices $v$ and $w$ remain such that $\omega(v) + \omega(w) \leq h$ and $\gamma(v) + \gamma(w) \leq \sqrt{h}$. Let $H'$ be the graph obtained when no more edge contractions are possible. We call a vertex $w \in H'$ heavy if $\omega(w) \geq h/2$ and $\gamma(w) \geq \sqrt{h}/2$, and light otherwise. We merge pairs of light vertices of degree two which are adjacent to the same two heavy neighbors until no two such vertices $v$ and $w$ remain. Let $H''$ be the graph obtained when no more light vertices can be merged. Every vertex $w_i \in H''$ corresponds to a subgraph $G_i$ of $G' - S$ which consists of the connected components $Q_{i_1}, \ldots, Q_{i_s}$ represented by the vertices $v_{i_1}, \ldots, v_{i_s}$ that have been merged to produce $w_i$. Let $\mathcal{P} = (S, \{G_1, \ldots, G_k\})$, where $k = |V(H'')|$. We claim that $k = \mathcal{O}(N/h)$ and that $\mathcal{P}$ is proper and regular.

To show the first claim, we argue as follows: We have already observed that $\sum_{i=1}^{r} |\partial Q_i| = \mathcal{O}\left(N/\sqrt{h}\right)$. This implies that $\sum_{i=1}^{r} \gamma(v_i) = \mathcal{O}\left(N/\sqrt{h}\right)$. Furthermore, $\sum_{i=1}^{r} \omega(v_i) \leq N$. Hence, graph $H''$ contains $\mathcal{O}(N/h)$ heavy vertices. Graph $H''$ is planar, as $H$ is planar and the above transformations preserve planarity. Hence, by Corollary 8.1, graph $H''$ contains $\mathcal{O}(N/h)$ light vertices. Every vertex in $H''$ gives rise to one subgraph $G_i$ in $\mathcal{P}$, so that $k = \mathcal{O}(N/h)$. The properness of $\mathcal{P}$ now follows because our construction explicitly ensures that no graph $G_i$ has size exceeding $h$ or boundary size more than $\sqrt{h}$

In order to prove that $\mathcal{P}$ is regular, we analyze the two stages of the computation of $H''$ from $H$. The construction of $H'$ from $H$ merges adjacent vertices. Two vertices in $H$ are adjacent only if their corresponding connected components share a boundary vertex. Hence, for every subgraphs $Q'$ corresponding to a vertex in $H'$, graph $R' = G[V(Q') \cup \partial Q']$ is connected. Since the construction of $H''$ from $H'$ merges only light vertices, the subgraphs $R'$ corresponding to heavy vertices in $H''$ remain

connected. The subgraphs corresponding to light vertices in $H'$ are also connected. Thus, disconnected subgraphs can be produced only by merging light vertices that are adjacent to the same set of at most two heavy neighbors. Hence, each of the subgraphs resulting from merging light vertices during the construction of $H''$ from $H'$ shares separator vertices with at most two other subgraphs $R_1$ and $R_2$, which correspond to heavy vertices in $H''$ and are thus connected.

The construction of graph $H$ from graph $G'$ and separator $S$ as well as the extraction of subgraphs $G_1, \ldots, G_k$ corresponding to the vertices in $H''$ can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os, as described in the proof of Lemma 8.14. Once graph $H$ is given, graph $H'$ can be constructed from graph $H$ using procedure ContractEdges (Algorithm 8.4), which takes $\mathcal{O}(\text{sort}(N))$ I/Os, by Lemma 8.6. The construction of graph $H''$ from graph $H'$ can be carried out using procedure MergeLowDegree (Algorithm 8.3), which takes $\mathcal{O}(\text{sort}(N))$ I/Os, by Lemma 8.4.

As both conditions of Theorem 10.1 can be satisfied in $\mathcal{O}(\text{sort}(N))$ I/Os, we obtain the following result.

**Theorem 10.2** *The single-source shortest path problem on a directed planar graph with $N$ vertices and non-negative edge weights can be solved in $\mathcal{O}(\text{sort}(N))$ I/Os using linear space, provided that $M \geq (DB)^2 \log^2(DB)$.*

**Corollary 10.1** *A BFS-tree of a directed planar graph with $N$ vertices can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os using linear space, provided that $M \geq (DB)^2 \log^2(DB)$.*

## 10.2 Separators of Low Cost and Edge Separators

In this section, we present an algorithm to compute an $\varepsilon$-separator of low cost for a planar graph whose vertices have been assigned non-negative costs and weights. In particular, we make use of the following result and show that the algorithm which computes the separator in the theorem can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os, using results from Chapter 9, Section 10.1, and [177].

**Theorem 10.3 (Aleksandrov et al. [9])** *Given a planar graph $G = (V, E)$, a cost function $\gamma : V \to \mathbb{R}^+$, a weight function $\omega : V \to \mathbb{R}^+$, and a real number $0 < \varepsilon < 1$, there exists an $\varepsilon$-separator $S$ of cost $\gamma(S) \leq 4\sqrt{2\sigma(G)/\varepsilon}$ for $G$, where $\sigma(G) = \sum_{v \in V}(\gamma(v))^2$. Such a separator can be computed in linear time.*

### 10.2.1 Outline of the Algorithm

Before describing an I/O-efficient algorithm for computing a separator as in Theorem 10.3, we recall the internal memory algorithm of [9], as the I/O-efficient algorithm is based on it: The algorithm first computes a planar embedding $\hat{G}$ of $G$ and triangulates the faces of $\hat{G}$. Then a vertex $\rho$ of weight and cost zero is introduced into one of the faces of $\hat{G}$. The algorithm connects this vertex to every vertex on the boundary of that face. From now on we use $G$ to refer to the resulting triangulation, since a separator of the triangulation is also a separator of the given graph $G$. Now $G$ is transformed into a directed graph by replacing every undirected edge in $G$ with its two corresponding directed edges. The weight of an edge is the cost of its target vertex. Given this weight function $\omega' : E \to \mathbb{R}_0^+$, the algorithm computes a shortest path tree $T$ of $G$ rooted at vertex $\rho$. Using tree $T$, separator $S$ is now computed in two phases.

The first phase cuts $T$ into shallow layers. In particular, it computes a separator $S_1 = L(x_1) \cup \cdots \cup L(x_p)$, where the vertices in a set $L(x_i)$ are at approximately the same distance from $\rho$. The vertices in a set $L(x_i)$ separate all vertices in $G$ that are closer to $\rho$ than the vertices in $L(x_i)$ from those that are further away. Thus, graph $G - S_1$ can be partitioned into subgraphs $G_0, \ldots, G_p$, so that graph $G_0$ contains all vertices above $L(x_1)$ in $T$, graph $G_p$ contains all vertices below $L(x_p)$ in $T$, and graph $G_i$, $0 < i < p$, contains all vertices between $L(x_i)$ and $L(x_{i+1})$.

The second phase partitions each graph $G_i$ whose weight exceeds $\varepsilon\omega(G)$ into subgraphs of weight at most $\varepsilon\omega(G)$. The separator used to obtain this partition consists of a collection of fundamental cycles of a shortest path tree $T_i$ of $G_i$. Since the layers of $T$ computed in the previous phase are shallow, tree $T_i$ has a small radius if rooted

at an appropriate vertex. This guarantees that the computed separator of $G_i$ is of low cost. Let $S_2$ be the set of separator vertices computed in this second phase. Then $S = S_1 \cup S_2$ is an $\varepsilon$-separator of $G$. Next we discuss the two phases of the algorithm in detail.

**Cutting $T$ into layers.** For every vertex $v \in G$, let $d(v) = \text{dist}_T(\rho, v)$. Let $r(T) = \max\{d(v) : v \in V\}$. We call $r(T)$ the *radius* of $T$. For every real number $x \in [0, r(T)]$, let $L(x) = \{w : e = (v, w) \in T \text{ and } d(v) < x \leq d(w)\}$. We denote the vertices of $G$ by $\rho = v_0, v_1, \ldots, v_N$ so that $0 = d(v_0) < d(v_1) < d(v_2) < \cdots < d(v_N) = r(T)$. Then $L(x) = L(y)$, for all $x, y \in (d(v_{i-1}), d(v_i)]$. Thus, there are only $N$ different sets $L(x)$. Every set $L(x)$ is a separator of $G$ which partitions the vertex set $V$ into two sets $V^-(x) = \{v \in V : d(v) < x\}$ and $V^+(x) = \{v \in V : d(v) \geq x \text{ and } v \notin L(x)\}$ so that no vertex in $V^-(x)$ is adjacent to a vertex in $V^+(x)$.

Now let $h = \sqrt{\varepsilon\sigma(G)/8}$, $p = \lfloor r(T)/h \rfloor$, and $y_i = ih$, for $0 \leq i \leq p$. For $1 \leq i \leq p$, let $V_i$ be the set of vertices $v \in V$ such that $d(v) \in (y_{i-1}, y_i]$. Let $w_i \in V_i$ be the vertex so that $\gamma(L(d(w_i)))$ is minimized, and let $x_i = d(w_i)$. Separator $S_1$ is defined as the set $S_1 = L(x_1) \cup \cdots \cup L(x_p)$. It is shown in [9] that $\gamma(S_1) \leq \sigma(G)/h = 2\sqrt{2\sigma(G)/\varepsilon}$. The removal of the vertices in $S_1$ partitions $G$ into subgraphs $G_0, \ldots, G_p$, where $G_i = G[V^+(x_{i-1}) \cap V^-(x_{i+1})]$.

**Partitioning the layers.** A subgraph $G_i$ whose weight exceeds $\varepsilon\omega(G)$ is partitioned into subgraphs of weight at most $\varepsilon\omega(G)$ as follows: If $i = 0$, graph $G_0$ is triangulated. Let $\rho_0 = \rho$. Otherwise, a new vertex $\rho_i$ of zero cost and weight is introduced and connected to all vertices in $G_i$ which are adjacent to vertices in $L(x_i)$. Then the algorithm triangulates the resulting graph and computes a shortest path tree $T_i$ of $G_i$ rooted at vertex $\rho_i$. A *fundamental cycle* of a non-tree edge $e$ is defined as the graph $C(e)$ containing edge $e$ as well as the unique path in $T_i$ between the endpoints of edge $e$. From the choice of levels $L(x_1), \ldots, L(x_p)$ it follows that $r(T_i) \leq 2h$, so that every fundamental cycle of $T_i$ has cost at most $4h$. On the other hand, $\left\lfloor \frac{2\omega(G_i)}{\varepsilon\omega(G)} \right\rfloor$ fundamental cycles are sufficient to partition $G_i$ into subgraphs of weight at most $\varepsilon\omega(G)$,

so that the separator used to partition $G_i$ has cost at most $4h\left\lfloor\frac{2\omega(G_i)}{\varepsilon\omega(G)}\right\rfloor$. Summing over all graphs $G_0,\ldots,G_p$, the cost of the set $S_2$ of separator vertices introduced in this second phase is $\gamma(S_2) \leq 8h/\varepsilon = 2\sqrt{2\sigma(G)/\varepsilon}$. Hence, the weight of separator $S = S_1 \cup S_2$ is $\gamma(S) \leq 4\sqrt{2\sigma(G)/\varepsilon}$. It remains to discuss the computation of $\left\lfloor\frac{2\omega(G_i)}{\varepsilon\omega(G)}\right\rfloor$ fundamental cycles which partition $G_i$ into subgraphs of weight at most $\varepsilon\omega(G)$. In order to compute these fundamental cycles, the idea of a *separation tree* is used, which has been introduced in [8]. The separation tree $Q_i$ of $G_i$ w.r.t. $T_i$ is defined as follows: Let $G_i^*$ be the dual of $G_i$. Then $Q_i$ is obtained from $G_i^*$ by removing all edges which are dual to edges in $T_i$. An edge-separator $X = \{e_1,\ldots,e_k\}$ of $Q_i$ defines a set of fundamental cycles $\mathcal{C}(X) = \{C(e_1^*),\ldots,C(e_k^*)\}$. Next we present an assignment of weights to the edges of graph $Q_i$, as proposed in [8], so that the set of fundamental cycles corresponding to an $\varepsilon$-edge separator of $Q_i$ is an $\varepsilon$-separator of $G_i$.

For every vertex $v \in V$, we define two sets of edges: The set $E_0(v)$ contains all non-tree edges incident to $v$. The set $E_1(v)$ contains all non-tree edges both of whose endpoints are adjacent to $v$. It is shown in [8] that $E_1(v) \neq \emptyset$, for all $v \in V$. Initially, let $\omega^*(e) = 0$, for every edge $e \in Q_i$. For every vertex $v \in V$ with $E_0(v) \neq \emptyset$, weight $\omega^*(e)$ is increased by $\omega(v)$, where $e^*$ is an arbitrary edge in $E_0(v)$. For every vertex $v \in V$ with $E_0(v) = \emptyset$, let $e^* \in E_1(v)$. Then weight $\omega^*(e)$ is increased by $\omega(v)$. It is shown in [8] that the set $\mathcal{C}(X)$ of fundamental cycles defined by an $\varepsilon$-edge separator $X$ of $Q_i$ is an $\varepsilon$-separator of $G_i$.

In order to compute a set $X$ of size at most $\left\lfloor\frac{2\omega(G_i)}{\varepsilon\omega(G)}\right\rfloor$, the algorithm of [8] chooses some vertex of $Q_i$ as the root and then processes $Q_i$ bottom-up. If an edge $e$ has weight exceeding $\frac{\varepsilon\omega(G)}{2\omega(G_i)}$, it is added to the separator $X$. Otherwise, the weight of its parent edge in $Q_i$ is increased by $\omega^*(e)$.

### 10.2.2 An I/O-Efficient Algorithm

Given the above algorithm for computing an $\varepsilon$-separator of low cost, we have to show that each of its steps can be carried out in an I/O-efficient manner.

**Computing $T$.** A planar embedding $\hat{G}$ of $G$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os using Theorem 9.2. In [177], it is shown how to triangulate an embedded planar graph in $\mathcal{O}(\text{sort}(N))$ I/Os. A representation of the faces of the resulting triangulation as lists of vertices on the boundary of each face can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os [177]. Given this representation, we scan one such list and make the new vertex $\rho$ adjacent to all vertices in this list. We use procedure CopyVertexLabels to inform every edge of $G$ about the cost of its endpoints. Then a single scan of the edge set of $G$ is sufficient to replace every edge of $G$ with its two corresponding directed edges and assign weights as defined above to these edges. The shortest path tree $T$ can now be computed in $\mathcal{O}(\text{sort}(N))$ I/Os using Theorem 10.2. Hence, tree $T$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, given only graph $G$ and an assignment of costs and weights to its vertices.

**Cutting $T$ into layers.** Given the distances of the vertices of $T$ from $\rho$, we compute values $x_1, \ldots, x_p$ as defined in Section 10.2.1 as follows: We inform the edges of $T$ about the costs and distances of their endpoints from the root $\rho$ of $T$ using procedure CopyVertexLabels and sort them by the distances of their source vertices from $\rho$. Now we scan the sorted edge list to simulate a sweep from $x = -\infty$ to $x = +\infty$. During the sweep, we maintain the current weight of set $L(x)$, the minimum weight $\gamma_{\min}(i)$ found in interval $(y_{i-1}, y_i]$, and the value $x_i \in (y_{i-1}, y_i]$ so that $\gamma(L(x_i)) = \gamma_{\min}(i)$. When $x = d(v_j)$, for some vertex $v_j$, we perform the following operations: If $d(v_{j+1}) > y_i$, we report $x_i$ and increase $i$ by one. Then we decrease $\gamma(L(x))$ by $\gamma(v_j)$ and increase $\gamma(L(x))$ by the total cost of the target vertices of all edges having $v_j$ as a source vertex. This produces $\gamma(L(d(v_{j+1})))$. If $\gamma(L(d(v_{j+1}))) < \gamma_{\min}(i)$, let $\gamma_{\min}(i) = \gamma(L(d(v_{j+1})))$ and $x_i = d(v_{i+1})$.

Given values $x_1 < \cdots < x_p$ and the edge set of $T$ as sorted above, we extract the set $S_1 = L(x_1) \cup \cdots \cup L(x_p)$ as follows: We scan the list of values $x_1, \ldots, x_p$ and the edge set of $T$, again to simulate a sweep from $x = -\infty$ to $x = +\infty$. When the sweep passes the left endpoint of an edge $e$, we append edge $e$ to a list $L$ of edges which contains all edges spanning the current sweep value $x$, but may contain more edges. When the

sweep passes a value $x_i$, we scan list $L$ to add all target vertices of edges $(v, w) \in L$, $d(v) < x_i \leq d(w)$, to $S_1$. After reporting these edges, we set $L = \emptyset$. The reason for resetting $L$ is that endpoints of edges that span two values $x_i$ and $x_{i+1}$ should not be reported more than once, and edges whose endpoints that have not been reported for $x_i$ cannot be reported for $x_{i+1}$, as for each such edge $(v, w)$, $d(w) < x_i < x_{i+1}$. As this phase requires on application of procedure CopyVertexLabels, sorting the edge set of $G$, and a constant number of scans, its I/O-complexity is $\mathcal{O}(\text{sort}(N))$.

**Partitioning the layers.** This phase of the algorithm extracts graphs $G_0, \ldots, G_p$, computes shortest path trees $T_0, \ldots, T_p$, and partitions each graph $G_i$, $0 \leq i \leq p$, using fundamental cycles of $T_i$.

We compute graphs $G_0, \ldots, G_p$ as follows: We apply procedure SetDifference to compute the set $V - S_1$. Then we sort the vertices in $V - S_1$ by their distances from $\rho$. Now a single scan of set $V - S_1$ and the list of values $0 = x_0, x_1, \ldots, x_p, x_{p+1} = r(T)$ is sufficient to partition $V - S_1$ into sets $V_0, \ldots, V_p$, where $x_i < d(v) \leq x_{i+1}$, for every vertex $v \in V_i$. We use procedure CopyVertexLabels to inform every edge about the sets $V_i$ and $V_j$ containing its endpoints. Now we sort the edges of $G$ to obtain a partition of $E$ into sets $E_0, \ldots, E_p$, $E'_1, \ldots, E'_p$, and $E''$. For all edges $(v, w) \in E_i$, $v, w \in V_i$. Set $E'_i$ contains all edges $(v, w)$, $d(v) < d(w)$, $v \in S_1$, and $w \in V_i$. Set $E''$ contains all edges which are not in any other set. We initialize graph $G_i$ as $G_i = (V_i, E_i)$, for all $0 \leq i \leq p$. Graph $G_0$ already contains vertex $\rho_0 = \rho$. Thus, it only has to be triangulated, using the algorithm of [177]. To prepare each graph $G_i$, $1 \leq i \leq p$, we add a new vertex $\rho_i$ to $V_i$, add edges $(\rho_i, v)$ and $(v, \rho_i)$ to $E_i$, for every edge $(u, v) \in E'_1$, and triangulate the resulting graph. This procedure requires the application of operations SetDifference and CopyVertexLabel and sorting and scanning sets of size $\mathcal{O}(N)$ a constant number of times. Hence, the extraction of graphs $G_0, \ldots, G_p$ takes $\mathcal{O}(\text{sort}(N))$ I/Os.

To partition graph $G_i$, we proceed as follows: We apply Theorem 10.2 to compute a shortest path tree $T_i$ of $G_i$ rooted at $\rho_i$. Then we compute the dual $G_i^*$ of $\hat{G}_i$ using an algorithm of [177]. We scan the edge list of $T_i$ and generate a list $D$ of

edges $e \in G_i^*$, $e^* \in T_i$, to be removed from $G_i^*$ in order to produce the separation tree $Q_i$. The removal of the edges in $D$ from $E_i^*$ can now be carried out using procedure SETDIFFERENCE. In total, the computation of trees $T_i$ and $Q_i$ takes $\mathcal{O}(\text{sort}(|G_i|))$ I/Os.

We compute weights $\omega^*(e)$, $e \in Q_i$, as follows: For every non-tree edge $e = \{v, w\} \in G_i$, we create two triples $(v, w, e^*)$ and $(w, v, e^*)$. We sort the list of these triples lexicographically, and sort the vertex set of $G_i$. Now a scan of these two lists suffices to find for every vertex $v \in G_i$ with $E_0(v) \neq \emptyset$, an edge $e \in Q_i$ to be charged for $\omega(v)$. We store this information in a list $X$ containing pairs $(e, \omega(v))$. During this scan, we copy all vertices which have no incident non-tree edge to a new list $V'$. For each such vertex with neighbors $w_1, \ldots, w_k$ in $T_i$, we create a list of pairs $(w_1, v), \ldots, (w_k, v)$. We sort the resulting list lexicographically. Now a single scan of this list and the sorted list of non-tree edges is sufficient to find for every vertex in $V'$, the set $E_1(v)$ of edges $e \in Q_i$ that can be charged for $\omega(v)$. In particular, this scan produces a set of pairs $(v, e)$, where $e \in E_1(v)$. We sort this set so that the pairs representing each set $E_1(v)$ are stored consecutively and scan it to choose the first edge $e$ from each set $E_1(v)$. For each such edge $e$, we add a pair $(e, \omega(v))$ to list $X$. Now we sort and scan list $X$ to compute for every edge $e \in Q_i$, its weight $\omega^*(e) = \sum_{j=1}^{k} \omega_j$, where $(e, \omega_1), \ldots, (e, \omega_k)$ are the entries in $X$ corresponding to edge $e$. This procedure involves sorting and scanning sets of size $\mathcal{O}(|G_i|)$ a constant number of times, so that the computation of weights $\omega^*(e)$, $e \in Q_i$ takes $\mathcal{O}(\text{sort}(|G_i|))$ I/Os.

To obtain an edge separator of $Q_i$, we apply the Euler tour technique and list-ranking to tree $Q_i$ to root it at an arbitrary vertex, compute a preorder numbering of $Q_i$, and direct all edges in $Q_i$ from children to parents. Then we apply time-forward processing to carry out the procedure of [8] for partitioning $Q_i$ into subgraphs of weight at most $\varepsilon$. Given the edge separator $X$ of $Q_i$ produced by this procedure, we apply procedure SUMEDGELABELS to mark the endpoints of all dual edges $e^* \in G_i$, $e \in X$, in $T_i$. We process $T_i$ bottom-up to find all vertices that belong to the fundamental cycles defined by $X$ and add these vertices to $S_2$. As

applying the Euler-tour technique, list-ranking, and time-forward processing to $Q_i$ take $\mathcal{O}(\text{sort}(|Q_i|)) = \mathcal{O}(\text{sort}(|G_i|))$ I/Os, the computation of the edge separator $X$ takes $\mathcal{O}(\text{sort}(|G_i|))$ I/Os. The application of procedure SUMEDGELABELS to the set of edges dual to the edges in $X$ and the vertex set of $G_i$ takes $\mathcal{O}(\text{sort}(|G_i|))$ I/Os. The final application of time-forward processing to extract the vertex set of the fundamental cycles in $\mathcal{C}(X)$ takes $\mathcal{O}(\text{sort}(|G_i|))$ I/Os.

We have shown that the partition of graph $G$ into graphs $G_0, \ldots, G_p$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os and that the partition of each graph $G_i$, $0 \leq i \leq p$, into subgraphs of weight at most $\varepsilon\omega(G)$ using fundamental cycles can be computed in $\mathcal{O}(\text{sort}(|G_i|))$ I/Os. Thus, the whole algorithm takes $\mathcal{O}\left(\text{sort}(N) + \sum_{i=0}^{p} \text{sort}(|G_i|)\right) = \mathcal{O}(\text{sort}(N))$ I/Os, and we obtain the following result.

**Theorem 10.4** *Given a planar graph $G = (V, E)$, a cost function $\gamma : V \to \mathbb{R}^+$, and a weight function $\omega : V \to \mathbb{R}^+$, a separator $S$ as in Theorem 10.3 can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space, provided that $M \geq (DB)^2 \log^2(DB)$.*

It is shown in [9] that Theorem 10.3 can also be used to compute optimal weighted edge-separators of planar graphs: We let the cost of a vertex be equal to its degree. Then we compute a vertex-separator $S$ of low cost. For every vertex $v \in S$, we add all edges incident to $v$ to the edge-separator $X$. It is easy to verify that the computation of the vertex costs and the extraction of set $X$ from set $S$ can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os. Hence, we obtain the following corollary.

**Corollary 10.2** *Let $G = (V, E)$ be a planar graph, $0 < \varepsilon < 1$ be a real number, and $\omega : V \to \mathbb{R}^+$ be a weight function so that $\omega(v) \leq \varepsilon\omega(G)$, for all $v \in V$. Then there exists a set $X$ of at most $4\sqrt{2\left(\sum_{v \in V} \deg^2(v)\right)/\varepsilon}$ edges so that no connected component of graph $(V, E \setminus X)$ has weight exceeding $\varepsilon\omega(G)$. Such an edge-separator $X$ can be found in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space, provided that $M \geq (DB)^2 \log^2(DB)$.*

# Chapter 11

# Depth-First Search in Planar Graphs

In this chapter, we apply the embedding algorithm of Chapter 9 and the BFS algorithm of Section 10.1 to obtain an $\mathcal{O}(\text{sort}(N))$ I/O algorithm for computing a DFS-tree of a connected planar graph. The algorithm is similar to that of Hagerup [92]; but the details differ. In particular, our algorithm appears to be simpler.

Since a DFS-tree of an arbitrary connected graph can easily be obtained from DFS-trees of its biconnected components, the difficult part of the algorithm is the computation of a DFS-tree of a biconnected planar graph. Given a planar embedding $\hat{G}$ of a biconnected planar graph $G$, we use BFS to partition the faces of $\hat{G}$ into levels around a source face $s$ which has the root $r$ of the DFS-tree on its boundary. We grow the DFS-tree level by level around face $s$. In Section 11.1, we formally define the levels into which the faces of $\hat{G}$ are partitioned, and we show how to obtain this partition in an I/O-efficient manner. We also show that this partition of the faces of $\hat{G}$ into levels induces a partition of $G$ into subgraphs $H_1, \ldots, H_k$ of a sufficiently simple structure that allows us to perform DFS in these subgraphs I/O-efficiently. These subgraphs have the additional property that a DFS-tree of $G$ can be obtained by combining appropriate DFS-forests of graphs $H_1, \ldots, H_k$. In Section 11.2, we study the structure of graphs $H_1, \ldots, H_k$ and show how to perform DFS in these graphs I/O-efficiently. In Sections 11.3 and 11.4, we show how to combine DFS-forests of

graphs $H_1, \ldots, H_k$ to obtain a DFS-tree of $G$ and how to obtain a DFS-tree of a connected planar graph from DFS-trees of its bicomps.

## 11.1 A Partition of the Graph into Layers

The goal of this section is to obtain a partition of a biconnected planar graph $G$ into *layers* $H_1, \ldots, H_k$. Each such layer $H_i$ is induced by the faces of a planar embedding $\hat{G}$ of $G$ that belong to *level* $i$ according to the following partition of the faces of $\hat{G}$ into levels:

Let $s$ be a face of $\hat{G}$ which has the root vertex $r$ of the DFS-tree on its boundary. Face $s$ is the only face at level 0. The levels of all other faces are defined inductively. A face $f$ is at level $i$ if it shares a vertex with a face at level $i-1$, but not with a face at level less than $i-1$ (see Figure 11.1a).

In order to obtain this partition of the faces of $G$ into levels around a source face, we apply BFS to the face-on-vertex graph $G_F$ of $G$ (see Figure 11.1b). In particular, the level $\ell(f)$ of a face $f \in \hat{G}$ is computed as $\ell(f) = d_{G_F}(s^*, f^*)/2$, where $d_{G_F}(s^*, f^*)$ denotes the depth of vertex $f^*$ in a BFS-tree of $G_F$ rooted at $s^*$.

Given the above partition of the faces of $G$ around face $s$, let $G_i$ be the subgraph of $G$ defined as the union of the boundaries of faces at level at most $i$, and let $H_i' = G_i - G_{i-1}$, for $i > 0$. Let $A_i$ be the set of edges in $H_i'$ that have exactly one endpoint in $G_{i-1}$. We call $A_i$ the set of *attachment edges* of graph $H_i = H_i' - A_i$. For $i = 0$, we define $H_0 = G_0$. We call graph $H_i$ the *$i$-th layer* of $G$. (See Figure 11.1c.) The vertices and edges of $H_i$ are said to be at *level* $i$. The edge sets of $G_{i-1}$ and $H_i$ together with the set $A_i$ of attachment edges of $H_i$ form a partition of the edge set of $G_i$. Algorithm 11.1 computes layers $H_0, \ldots, H_k$ as well as all sets $A_1, \ldots, A_k$ of attachment edges. The following lemma shows that this algorithm is correct and takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os to obtain this partition of graph $G$.

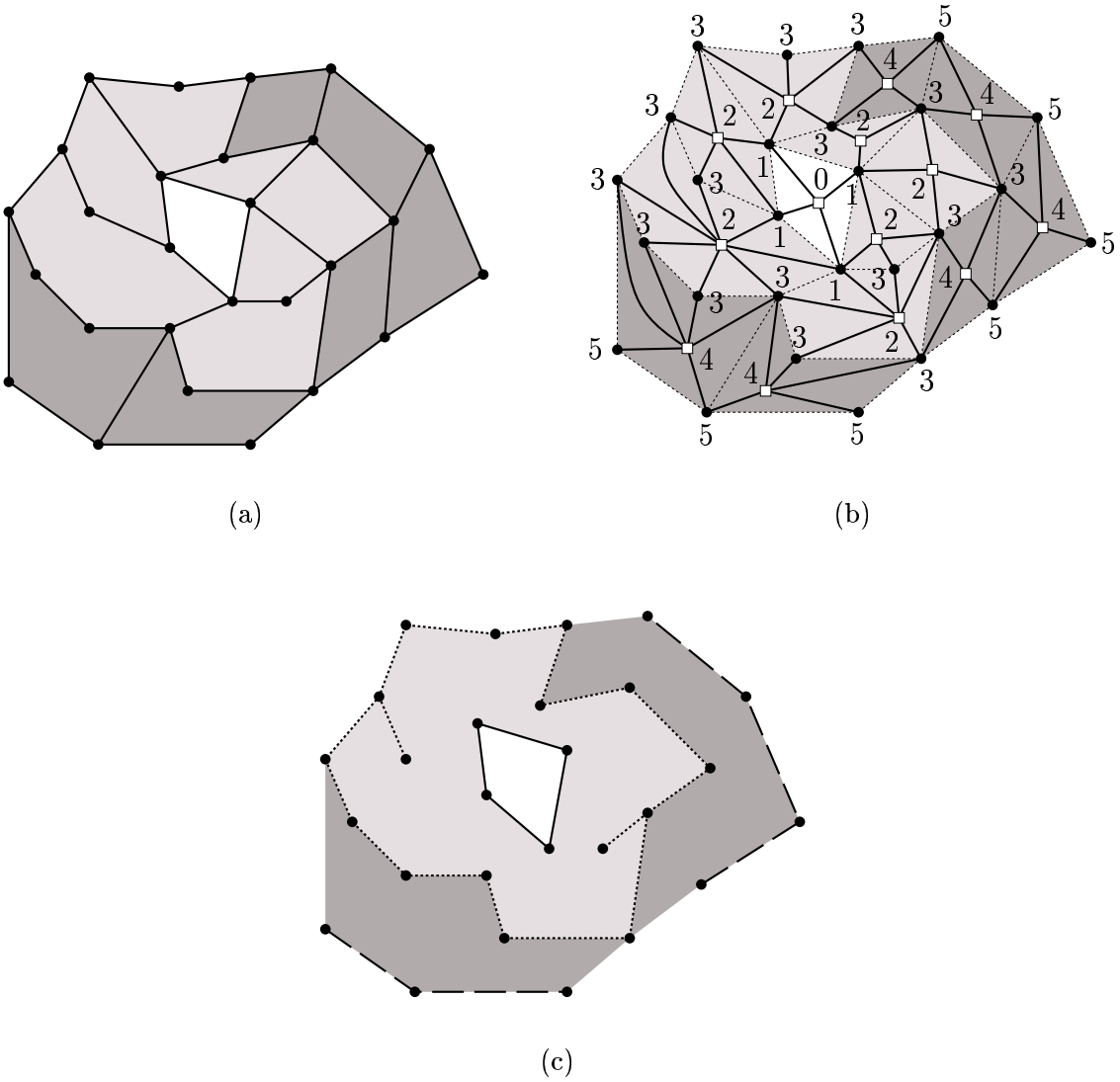



**Figure 11.1**
(a) A planar embedding $\hat{G}$ of a planar graph $G$ with its faces colored according to their levels. The level-0 face $s$ is white. Level-1 faces are light gray. Level-2 faces are dark gray. (b) The face-on-vertex graph $G_F$ of $G$. Vertices are labelled with their distances from the source vertex $s^*$. (c) The layers of $G$. Layer 0 is solid. Layer 1 is dotted. Layer 2 is dashed.

**Procedure** ComputeLayers

**Input:** A biconnected planar graph $G = (V, E)$ and a vertex $r \in V$.
**Output:** A partition of $G$ into layers $H_0, \ldots, H_k$ and sets $A_1, \ldots, A_k$ of attachment edges of layers $H_1, \ldots, H_k$.

1: Compute a planar embedding $\hat{G}$ of $G$.
2: Identify the faces of $\hat{G}$ and represent each such face $f$ as a list $V(f)$ of vertices clockwise along the boundary of face $f$.
3: Compute the face-on-vertex graph $G_F$ of $\hat{G}$.
4: Choose a face $s$ so that $r \in V(s)$.
5: Apply BFS in $G_F$ to compute $\ell(f)$, for all faces $f \in \hat{G}$.
6: **for** every edge $e = \{v, w\} \in E(G)$ **do**
7: Let $f_1$ and $f_2$ be the two faces of $\hat{G}$ so that $\{v, w\} \subseteq V(f_1) \cap V(f_2)$.
8: $\ell(e) \leftarrow \min(\ell(f_1), \ell(f_2))$
9: **end for**
10: **for** every vertex $v \in V(G)$ **do**
11: $\ell(v) \leftarrow \min\{\ell(\{v, w\}) : \{v, w\} \in E(G)\}$
12: **end for**
13: $k \leftarrow \max\{\ell(e) : e \in E(G)\}$
14: **for** $i = 0, \ldots, k$ **do**
15: $A_i \leftarrow \{\{v, w\} \in E(G) : \ell(v) = i \text{ and } \ell(w) < i\}$
16: $V(H_i) \leftarrow \{v \in V(G) : \ell(v) = i\}$
17: $E(H_i) \leftarrow \{\{v, w\} \in E(G) : \ell(v) = \ell(w) = i\}$
18: **end for**

**Algorithm 11.1**
Computing the layers of a biconnected planar graph $G$ around a source face $s$.

**Lemma 11.1** *Algorithm 11.1 takes $\mathcal{O}(\text{sort}(N))$ I/Os and uses linear space to compute layers $H_0, \ldots, H_k$ of graph $G$ together with the sets $A_1, \ldots, A_k$ of attachment edges of layers $H_1, \ldots, H_k$.*

*Proof.* The correctness of the algorithm is obvious. We show that Algorithm 11.1 takes $\mathcal{O}(\text{sort}(N))$ I/Os. Computing a planar embedding of $G$ in Line 1 of the algorithm takes $\mathcal{O}(\text{sort}(N))$ I/Os, by Theorem 9.2. It is shown in [177, page 110] how to compute a representation of the faces of $\hat{G}$ as vertex lists $V(f)$, sorted clockwise around each face $f$. The procedure of [177] takes $\mathcal{O}(\text{sort}(N))$ I/Os.

Given this information, the face-on-vertex graph $G_F$ of embedding $\hat{G}$ can be computed in $\mathcal{O}(\text{scan}(N))$ I/Os: We start with the graph $(V, \emptyset)$. Then we scan the lists $V(f)$, for all faces $f \in \hat{G}$. For every face $f \in \hat{G}$, we add a vertex $f^*$ to $V(G_F)$. For every vertex $v \in V(f)$, we add an edge $\{f^*, v\}$ to $E(G_F)$.

Line 4 takes a single scan over lists $V(f)$ and can in fact be incorporated in the computation of Line 3. The computation of levels in Line 5 takes $\mathcal{O}(\text{sort}(N))$ I/Os, by Corollary 10.1. In order to carry out Lines 6–9, we generate triples $(v, w, \ell(f))$, for all pairs of consecutive vertices in all lists $V(f)$. This takes $\mathcal{O}(\text{scan}(N))$ I/Os. Then we sort this list so that for every edge $\{v, w\}$, the two entries $(v, w, \ell(f_1))$ and $(v, w, \ell(f_2))$ are stored consecutively, where $f_1$ and $f_2$ are the two faces containing edge $e = \{v, w\}$. Now a single scan of this list is sufficient to compute a list of triples $(v, w, \ell(e) = \min(\ell(f_1), \ell(f_2)))$. Lines 10–12 can be carried out using procedure SumEdgeLabels. Line 13 is present only to clarify the description of the algorithm, as Lines 14–18 can be realized by sorting the vertex set of $G$ and sorting the edge set of $G$ lexicographically. □

## 11.2 Depth-First Search in a Layer

In this section, we show how to compute a DFS-forest $F_i$ for a single layer $H_i$ of $G$. Our procedure for constructing a DFS-tree of $G$, which we describe in Section 11.3, requires every vertex $v \in H_i$ to be labelled with its *DFS-depth* in forest $F_i$. This is the distance $d_{F_i}(r_{i,j}, v)$ in $F_i$, where $r_{i,j}$ is the root of the DFS-tree $T_{i,j}$ in $F_i$ containing $v$.

If $i = 0$, $H_i$ consists only of the boundary of face $s$. Hence, a DFS-tree $F_0$ of $H_0$ rooted at vertex $r_0 = r$ can be computed by removing one of the edges of $H_0$ incident to $r_0$ from $H_0$. Given this information, the Euler tour technique and list-ranking can be used to compute for every vertex $v \in H_0$, its DFS-depth $d_{F_0}(r_0, v)$, as $F_0$ is a simple path.

Before we describe our algorithm for computing a DFS-tree for a layer $H_i$, $i > 0$, we prove that these layers have a very simple structure. This is the key to obtaining an I/O-efficient DFS-algorithm for these graphs.

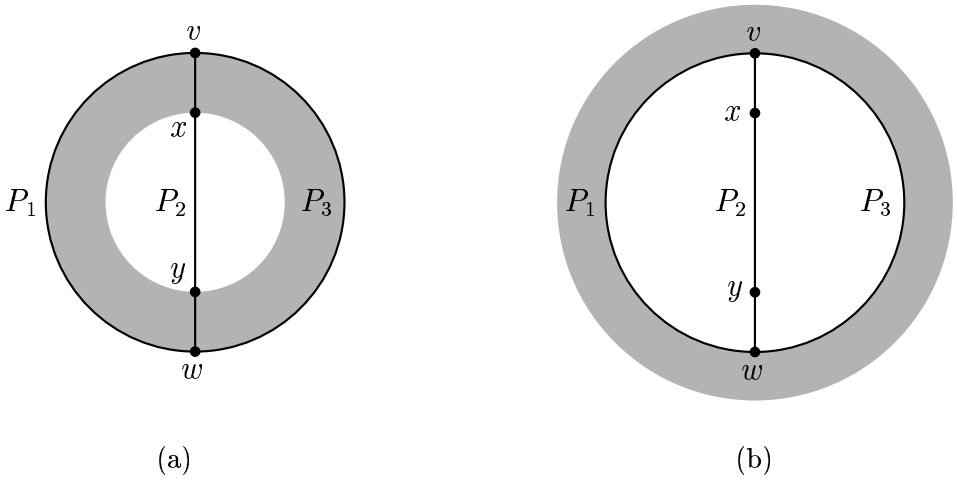



**Figure 11.2**
Proof of Lemma 11.2. Shaded faces are at level $i$.

**Lemma 11.2** *The non-trivial bicomps of $H_i$ are the boundary cycles of $G_i$.*

*Proof.* Consider a cycle $C$ in $H_i$. All faces incident to $C$ are at level $i$ or greater. Thus, since the subgraph of $G_F$ corresponding to the interior faces of $G_{i-1}$ is connected, all its faces are either inside or outside $C$. Assume w.l.o.g. that $G_{i-1}$ is inside $C$. Then none of the faces outside $C$ shares a vertex with a level-$(i-1)$ face. That is, all faces outside $C$ must be at level at least $i+1$, which means that $C$ is a boundary cycle of $G_i$.

Every bicomp that is not a cycle contains at least three internally vertex-disjoint paths $P_1$, $P_2$, and $P_3$ with the same endpoints $v$ and $w$ (see Figure 11.2). As we have just shown, the graph $C_1 = P_1 \cup P_3$ is a boundary cycle of $G_i$, as is the graph $C_2 = P_1 \cup P_2$. Let $\{v, x\}$ be the first edge of $P_2$, and $\{y, w\}$ be the last edge of $P_2$. Since $C_1$ is a boundary cycle of $G_i$, $G_i$ is either completely inside or completely outside $C_1$. Since $C_1$ is a subgraph of $H_i$, all faces incident to $C_1$ that are on the same side of $C_1$ as $G_i$ are at level $i$ because all faces on the other side of $C_1$ are at level at least $i+1$. Hence, if $P_2$ is on the same side of $C_1$ as $G_i$ (Figure 11.2a), the four faces incident to edges $\{v, x\}$ and $\{y, w\}$ are at level $i$, which contradicts the fact that $C_2$ is a boundary cycle of $G_i$. If $P_2$ is on the other side of $C_1$ (Figure 11.2b), the four

faces incident to edges $\{v, x\}$ and $\{y, w\}$ are at level at least $i + 1$, which contradicts the fact that edges $\{v, x\}$ and $\{y, w\}$ are at level $i$. Thus, every bicomp of $H_i$ consists of a single boundary cycle. □

Lemma 11.2 suggests the following strategy to compute a DFS-tree $T_{i,j}$ of a connected component $H_{i,j}$ of a layer $H_i$: First we compute the bicomp-cutpoint-tree $B_{i,j} = T_{\text{bic}}(H_{i,j})$ of $H_{i,j}$. Recall that this tree contains all cutpoints $v_1, \ldots, v_r$ as well as one vertex $\beta_h$ per bicomp $\mathcal{B}_h$ of $H_{i,j}$. There is an edge $\{v_q, \beta_h\}$ in $B_{i,j}$ if $v_q \in \mathcal{B}_h$. We choose a bicomp node $\beta_x$ as the root of $B_{i,j}$ so that $r_{i,j} \in \mathcal{B}_x$, where $r_{i,j}$ is the root of tree $T_{i,j}$. Then the parent cutpoint of a bicomp $\mathcal{B}_h \neq \mathcal{B}_x$ is the cutpoint $p(\beta_h)$, where $p(\beta_h)$ denotes the parent of node $\beta_h$ in $B_{i,j}$. For $\mathcal{B}_x$, we define $p(\beta_x) = r_{i,j}$. By Lemma 11.2, every bicomp $\mathcal{B}_h$ of $H_{i,j}$ consists either of a single edge or is a cycle. In the former case, we add $\mathcal{B}_h$ to $T_{i,j}$. In the latter case, we add all edges of $\mathcal{B}_h$ to $T_{i,j}$, except one of the edges incident to $p(\beta_h)$.

Before we show how to compute a DFS-forest $F_i$ as defined above in an I/O-efficient manner, we show that graph $T_{i,j}$ as defined above is indeed a DFS-tree of $H_{i,j}$.

**Lemma 11.3** *Graph $T_{i,j}$ is a DFS-tree of graph $H_{i,j}$.*

*Proof.* In order to prove the lemma, we have to show that $T_{i,j}$ is connected, contains a unique path between any two vertices $v$ and $w$, and for every edge $\{v, w\} \in H_{i,j} - T_{i,j}$, w.l.o.g. $v$ is an ancestor of $w$.

The connectivity of $T_{i,j}$ can be shown as follows: Graph $H_{i,j}$ is connected. Now let $v$ and $w$ be two arbitrary vertices in $H_{i,j}$, and let $P = (v = u_0, \ldots, u_z = w)$ be a path from $v$ to $w$ in $H_{i,j}$. If all edges in $P$ are in $T_{i,j}$, then $P$ is also a path from $v$ to $w$ in $T_{i,j}$. Otherwise, let $\{u_q, u_{q+1}\}$ be an edge that is not in $T_{i,j}$. Then edge $\{u_q, u_{q+1}\}$ is part of a cycle $\mathcal{B}_h = (u_q = x_1, \ldots, x_s = u_{q+1})$. Since at most one edge is removed from every bicomp $\mathcal{B}_h$, edge $\{u_q, u_{q+1}\} \in P$ can be replaced by the path $(u_q = x_1, \ldots, x_s = u_{q+1})$ in $T_{i,j}$. By replacing all edges in $P - T_{i,j}$ in this manner, we obtain a (not necessarily simple) path $P'$ from $v$ to $w$ in $T_{i,j}$.

Now let $v$ and $w$ be two vertices of $H_{i,j}$, and assume that $T_{i,j}$ contains two paths $P_1$ and $P_2$ from $v$ to $w$. Then $P_1$ and $P_2$ contain two internally vertex-disjoint subpaths $P_1'$ and $P_2'$ with the same endpoints $x$ and $y$. The graph $C = P_1' \cup P_2'$ is a cycle and hence, by Lemma 11.2, a non-trivial bicomp of $H_{i,j}$. But one edge is removed from every non-trivial bicomp of $H_{i,j}$ during the construction $T_{i,j}$. Hence, $C \not\subseteq T_{i,j}$, which contradicts the assumption that $P_1 \cup P_2 \subseteq T_{i,j}$.

To see that for every edge $\{v, w\} \in H_{i,j} - T_{i,j}$, $v$ is an ancestor of $w$, let $\mathcal{B}_h$ be the bicomp of $H_{i,j}$ containing edge $\{v, w\}$. Then w.l.o.g. $v = p(\beta_h)$, and every path from $r_{i,j}$ to $w$ contains $v$. Hence, the path from $r_{i,j}$ to $w$ in $T_{i,j}$ contains $v$, and $v$ is an ancestor of $w$ in $T_{i,j}$. □

The following corollary is an immediate consequence of Lemma 11.2 and the proof of Lemma 11.3.

**Corollary 11.1** *For every boundary cycle $C = (v_1, \ldots, v_x)$ of graph $G_i$, there is an index $1 \leq i \leq x$ such that*

*(i)* $(v_i, \ldots, v_x, v_1, \ldots, v_{i-1})$ *is a path in* $F_i$,

*(ii) For* $1 \leq j < i$, *vertex* $v_j$ *is an ancestor in* $F_i$ *of vertices* $v_{j+1}, \ldots, v_{i-1}$, *and*

*(iii) For* $i \leq j \leq x$, *vertex* $v_j$ *is an ancestor in* $F_i$ *of vertices* $v_{j+1}, \ldots, v_x, v_1, \ldots, v_{i-1}$.

We use Algorithm 11.2 to compute a DFS-forest $F_i$ of $H_i$. The following lemma shows that Algorithm 11.2 is I/O-efficient.

**Lemma 11.4** *Algorithm 11.2 takes $\mathcal{O}(\text{sort}(|H_i|))$ I/Os and uses linear space to compute a DFS-forest $F_i$ of layer $H_i$.*

*Proof.* The correctness of Algorithm 11.2 follows from Lemma 11.3 and the fact that the algorithm computes graph $F_i$ as defined in the text above. If we can show that the $j$-th iteration of the loop in Lines 1-12 takes $\mathcal{O}(\text{sort}(|H_{i,j}|))$ I/Os, the whole algorithm takes $\mathcal{O}(\text{sort}(|H_{i,1}|) + \cdots + \text{sort}(|H_{i,t}|)) = \mathcal{O}(\text{sort}(|H_i|))$ I/Os, as desired.

**Procedure** LayerDFS

**Input:** A layer $H_i$ of $G$ with connected components $H_{i,1}, \ldots, H_{i,t}$, and a set $r_{i,1}, \ldots, r_{i,t}$ of vertices such that $r_{i,j} \in H_{i,j}$, for all $1 \leq j \leq t$.

**Output:** A DFS-forest $F_i$ of $H_i$ consisting of DFS-trees $T_{i,1}, \ldots, T_{i,t}$ for the connected components of $H_i$. Tree $T_{i,j}$ is rooted at vertex $r_{i,j}$, for $1 \leq j \leq t$.

1: **for** $j = 1, \ldots, t$ **do**
2: $T_{i,j} \leftarrow H_{i,j}$
3: Compute the bicomps $\mathcal{B}_1, \ldots, \mathcal{B}_p$ of $H_{i,j}$.
4: Build the bicomp cutpoint tree $B_{i,j}$ of $H_{i,j}$.
5: Choose a root vertex $\beta_x$ of $B_{i,j}$ and determine the parent cutpoint $p(\beta_h)$, for every bicomp $\mathcal{B}_h$ of $H_{i,j}$.
6: **for** $h = 1, \ldots, p$ **do**
7: **if** $\mathcal{B}_h$ contains more than one edge **then**
8: Remove an edge in $\mathcal{B}_h$ incident to $p(\beta_h)$ from $T_{i,j}$.
9: **end if**
10: **end for**
11: Compute the DFS-depth $d_{T_{i,j}}(r_{i,j}, v)$, for every vertex $v \in H_{i,j}$.
12: **end for**
13: $F_i \leftarrow T_{i,1} \cup \cdots \cup T_{i,t}$

**Algorithm 11.2**
Computing a DFS-forest for a layer $H_i$ of $G$.

Computing the bicomps of $H_{i,j}$ in Line 3 takes $\mathcal{O}(\text{sort}(|H_{i,j}|))$ I/Os using an algorithm of [43]. Given that every edge is labelled as belonging to a particular bicomp, we sort the edges of $H_{i,j}$ by their bicomp labels and scan the resulting list to create one vertex $\beta_h$ per bicomp. In order to identify the cutpoints, we generate pairs $(v, \beta_h)$ and $(w, \beta_h)$, for every edge $\{v, w\} \in \mathcal{B}_h$. Then we sort the resulting list lexicographically. Now a single scan of this list suffices to determine for all vertices $v \in H_{i,j}$, whether the edges incident to $v$ are in more than one bicomp. If this is the case, $v$ is a cutpoint. We add $v$ to $V(B_{i,j})$ in this case. Also, for every cutpoint $v$, we keep one copy of every distinct entry $(v, \beta_h)$ as representing the edge $\{v, \beta_h\} \in B_{i,j}$. Hence, Line 4 takes $\mathcal{O}(\text{sort}(|H_{i,j}|))$ I/Os as well. In order to find a bicomp $\mathcal{B}_x$ containing $r_{i,j}$, we scan the edge list of $H_{i,j}$ until we find an edge

$e = \{r_{i,j}, v\}$. Then we choose $\mathcal{B}_x$ as the bicomp corresponding to the label of edge $e$. This computation can be incorporated in the computation of $B_{i,j}$. Rooting $B_{i,j}$ at vertex $\beta_h$ and computing the parent cutpoints $p(\beta_h)$, for all bicomps $\mathcal{B}_h$ of $H_{i,j}$, now takes $\mathcal{O}(\text{sort}(|H_{i,j}|))$ I/Os using the Euler tour technique and list-ranking [43]. Finally, to implement Lines 6–10, we sort the set of parent cutpoints $p(\beta_h)$ by their corresponding bicomps $\mathcal{B}_h$. We sort the edges of $H_{i,j}$ by their bicomp labels. Now a single scan of the two sorted lists of parent cutpoints and edges is sufficient to decide for every bicomp $\mathcal{B}_h$, whether it is non-trivial, and if so, remove an edge incident to $p(\beta_h)$ from the edge list. This finishes the computation of $T_{i,j}$. Given $T_{i,j}$ and $r_{i,j}$, the DFS-depths of all vertices in $H_{i,j}$ can now be computed in $\mathcal{O}(\text{sort}(|H_{i,j}|))$ I/Os using the Euler tour technique and list-ranking again [43]. □

## 11.3 Depth-First Search in a Biconnected Component

In order to compute a DFS-tree $T$ of a biconnected planar graph $G$, we partition $G$ into its layers. Then we compute an appropriate DFS-forest for each of the layers. Now we construct tree $T$ incrementally, starting with the DFS-tree $T_0 = F_0$ computed for layer $H_0 = G_0$. Given a DFS-tree $T_{i-1}$ of $G_{i-1}$, we obtain a DFS-tree of $G_i$ by attaching the trees in the DFS-forest of layer $H_i$ to $T_{i-1}$ using appropriate attachment edges of layer $H_i$. Algorithm 11.3 shows the details of this computation. Figure 11.3 illustrates the computation of Algorithm 11.3. Next we show that the algorithm produces a DFS-tree $T$ of $G$ and that it does so I/O-efficiently.

**Lemma 11.5** *The graph $T$ computed by Algorithm 11.3 for a biconnected planar graph $G = (V, E)$ is a DFS-tree of $G$.*

*Proof.* First we show that $T$ is a tree. In order to see that graph $T$ is connected, we argue inductively. After Line 3, $T$ is a spanning tree of $G_0 = H_0$. Hence, at the beginning of the $i$-th iteration of the loop in Lines 4–14, graph $T$ is a spanning graph for graph $G_{i-1}$. The $i$-th iteration of this loop computes spanning trees for the

**Procedure** BiconnectedDFS

**Input:** A biconnected planar graph $G = (V, E)$ and a vertex $r \in V$.
**Output:** A DFS-tree $T$ of $G$ rooted at vertex $r$.

1: Apply procedure ComputeLayers to partition $G$ into its layers $H_0, \ldots, H_k$ and sets $A_1, \ldots, A_k$ of attachment edges of layers $H_1, \ldots, H_k$.
2: Apply procedure LayerDFS to compute a DFS-tree $F_0$ of layer $H_0$ rooted at vertex $r$.
3: $T \leftarrow F_0$
4: **for** $i = 1, \ldots, k$ **do**
5: Compute the connected components $H_{i,1}, \ldots, H_{i,t}$ of $H_i$.
6: **for** $j = 1, \ldots, t$ **do**
7: Let $A_{i,j}$ be the set of attachment edges in $A_i$ that have an endpoint in $H_{i,j}$.
8: Let $e_{i,j} = \{v, w\}$, $v \in H_{i-1}, w \in H_i$, be the edge in $A_i$ so that the DFS-depth of $v$ in $F_{i-1}$ is maximized.
9: $T \leftarrow T \cup \{e_{i,j}\}$
10: Choose $r_{i,j} = w$ as the root of the DFS-tree $T_{i,j}$ to be computed for $H_{i,j}$.
11: **end for**
12: Apply procedure LayerDFS to compute a DFS-forest $F_i$ for layer $H_i$.
13: $T \leftarrow T \cup F_i$
14: **end for**

**Algorithm 11.3**
A DFS-algorithm for biconnected planar graphs.

connected components of $H_i$ and attaches each of them to $T$ using an attachment edge. Thus, at the end of the $i$-th iteration, graph $T$ is a spanning graph of $G_i$.

To see that graph $T$ is a tree, we have to show that graph $T$ contains $N - 1$ edges. After Line 3, $T$ contains $|G_0| - 1$ edges. Hence, at the beginning of the $i$-th iteration of the loop in Lines 4–14, graph $T$ contains $|G_{i-1}| - 1$ edges. For every connected component $H_{i,j}$ of $H_i$, we add $|H_{i,j}|$ edges to $T$: $|H_{i,j}| - 1$ edges in the spanning tree $T_{i,j}$ of $H_{i,j}$, and one attachment edge. Hence, at the end of the $i$-th iteration of the loop, graph $T$ contains $|G_i| - 1$ edges.

To finish the proof, we have to show that for every edge $\{v, w\} \in E(G) - E(T)$, w.l.o.g. $v$ is an ancestor of $w$ in $T$. If vertices $v$ and $w$ are in the same layer, then $v$ is an ancestor of $w$, by Lemma 11.3. Otherwise, $v \in H_{i-1}$ and $w \in H_i$. Let $C$

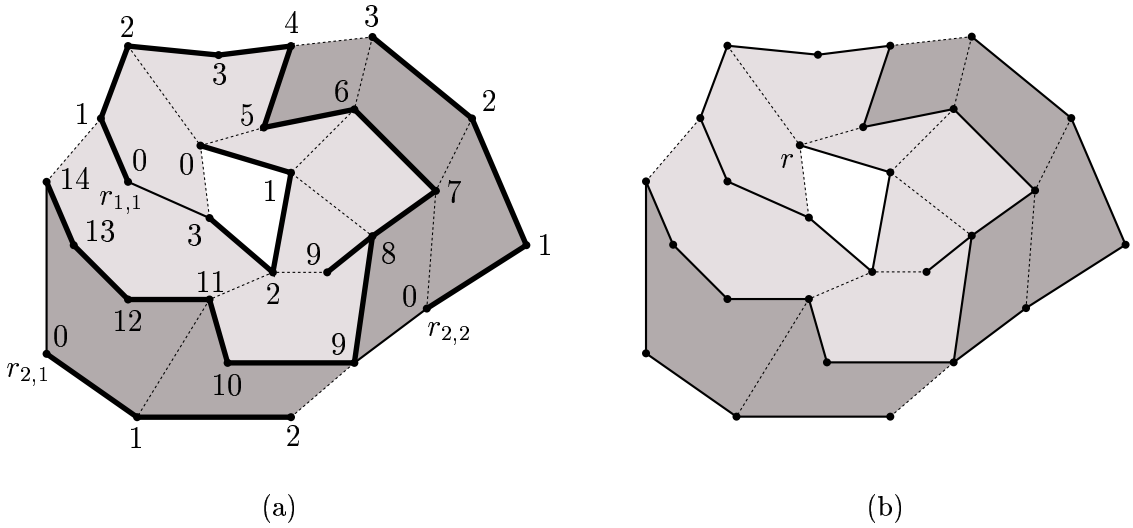



**Figure 11.3**
(a) DFS-forests of the layers of graph $G$ are shown in bold. Thin solid edges are the attachment edges used to join the DFS-forests of layers $H_0$, $H_1$, and $H_2$ in order to obtain a DFS-tree of $G$. Dotted edges are non-tree edges. The vertices in every layer are labelled with their DFS-depths. (b) The final DFS-tree of $G$.

be the boundary cycle of $G_{i-1}$ enclosing $w$. Then $v \in C$. Let $H_{i,j}$ be the connected component of $H_i$ containing $w$, and let $\{r_{i,j}, u\}$ be the attachment edge of $H_{i,j}$ in $T$. Then $u \in C$. Vertex $w$ is a descendant of $r_{i,j}$ and hence of $u$, and by the choice of edge $\{r_{i,j}, u\}$ and Corollary 11.1, $v$ is an ancestor of $u$. □

**Lemma 11.6** *Algorithm 11.3 takes $\mathcal{O}(\text{sort}(N))$ I/Os and uses linear space when applied to a biconnected planar graph $G = (V, E)$ with $N$ vertices.*

*Proof.* By Lemma 11.1, Line 1 of Algorithm 11.3 takes $\mathcal{O}(\text{sort}(N))$ I/Os. Line 2 takes $\mathcal{O}(\text{sort}(|H_0|))$ I/Os, by Lemma 11.4. We show that the $i$-th iteration of the loop in Lines 4–14 takes $\mathcal{O}(\text{sort}(|H_{i-1}| + |H_i|))$ I/Os, which implies that the total I/O-complexity of Lines 2–14 is $\mathcal{O}\left(\sum_{i=1}^{k} \text{sort}(|H_{i-1}| + |H_i|)\right) = \mathcal{O}(\text{sort}(N))$.

Computing the connected components of $H_i$ in Line 5 takes $\mathcal{O}(\text{sort}(|H_i|))$ I/Os using an algorithm of [43]. To find the root vertices $r_{i,1}, \ldots, r_{i,t}$ and attachment edges $e_{i,1}, \ldots, e_{i,t}$, for all connected components $H_{i,1}, \ldots, H_{i,t}$ of $H_i$, we apply procedure

CopyVertexLabels to inform every edge in $A_i$ about the DFS-depth of its endpoint in $H_{i-1}$ and the connected component $H_{i,j}$ of $H_i$ containing its other endpoint. This takes $\mathcal{O}(\text{sort}(|H_{i-1}| + |H_i|))$ I/Os, as $|A_i| = \mathcal{O}(|H_{i-1}| + |H_i|)$. Now we sort the edges in $A_i$ so that the edges in each set $A_{i,j}$ of attachment edges of connected component $H_{i,j}$ are stored consecutively. Then a single scan of $A_i$ is sufficient to find for every connected component $H_{i,j}$, the attachment edge $e_{i,j} = \{r_{i,j}, v\}$ so that the DFS-depth of $v$ is maximized. Again, this takes $\mathcal{O}(\text{sort}(|H_{i-1}| + |H_i|))$ I/Os. □

## 11.4 Depth-First Search in a Connected Planar Graph

In order to compute a DFS-tree of a connected planar graph $G$, we can use Algorithm 11.2. Only Lines 7–9 have to be replaced with a call to procedure BiconnectedDFS, which computes a DFS-tree of bicomp $\mathcal{B}_i$ rooted at vertex $p(\beta_i)$. The following theorem now follows from Lemmas 11.4 and 11.6.

**Theorem 11.1** *Given an undirected planar graph $G = (V, E)$, a DFS-tree of $G$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space, provided that $M \geq (DB)^2 \log^2(DB)$.*

# Part II

# Geometric Spanners and Proximity Problems

# Chapter 12

# The Well-Separated Pair Decomposition and Applications

In this chapter, we present an I/O-efficient algorithm for computing a well-separated pair decomposition of a point set in $d$-dimensional space. Roughly speaking, this is a partition of the point set into pairs of sets so that the two sets in each pair are far apart from each other compared to the sizes of their bounding rectangles. This partition can be used to obtain efficient solutions to a number of proximity and related problems.

An important and the most difficult step of the algorithm is the computation of a fair split tree of the point set. This tree and the subdivision it induces are used in Chapter 14 to obtain a planar Steiner spanner of a point set or set of polygonal obstacles in the plane.

In Section 12.1, we introduce the terminology used throughout Chapters 12–14. In Section 12.2, we introduce the topology buffer tree as an I/O-efficient data structure to answer search queries on static binary trees. This structure is used in Sections 12.3 and 13.3 to answer queries on the fair split tree of a point set. In Section 12.3, we present an I/O-efficient algorithm for constructing a fair split tree of a point set. In Section 12.4, we show how to derive a well-separated pair decomposition of a point set from its fair split tree. Finally, in Section 12.5, we present applications of

this data structure. In particular, we use it to develop I/O-efficient algorithms for computing a $t$-spanner of linear size and spanner diameter $\mathcal{O}(\log N)$ for a point set in $d$-dimensional space, and for solving the $K$-closest pair and $K$-nearest neighbor problems in $d$ dimensions. The dumbbell spanner discussed in Chapter 13 is based on the spanner presented in Section 12.5, so that the spanner presented here can be seen as a fundamental step for constructing a spanner which allows spanner paths to be reported I/O-efficiently.

## 12.1 Definitions

We use the following terminology throughout the remainder of this thesis. Most of these definitions are taken from [35], even though we have changed the definitions of concepts related to the fair split tree of a point set to suit our needs. For a given point set $S \subset \mathbb{R}^d$, the *bounding rectangle* $R(S)$ is the smallest rectangle containing all points in $S$, where a *rectangle* $R$ is the Cartesian product $[x_1, x_1'] \times [x_2, x_2'] \times \cdots \times [x_d, x_d']$ of a set of closed intervals. The *length* of $R$ in dimension $i$ is $\ell_i(R) = x_i' - x_i$. The maximum and minimum lengths of $R$ are $\ell_{\max}(R) = \max\{\ell_i(R) : 1 \leq i \leq d\}$ and $\ell_{\min}(R) = \min\{\ell_i(R) : 1 \leq i \leq d\}$. We call $R$ a *box* if $\ell_{\max}(R) \leq 3\ell_{\min}(R)$. If all lengths of $R$ are equal, $R$ is a *cube*. We denote its side length by $\ell(R) = \ell_{\max}(R) = \ell_{\min}(R)$. Let $i_{\max}(R)$ be a dimension so that $\ell_{i_{\max}(R)}(R) = \ell_{\max}(R)$ and $i_{\min}(R)$ be a dimension so that $\ell_{i_{\min}(R)}(R) = \ell_{\min}(R)$. For a point set $S$, let $\ell_i(S) = \ell_i(R(S))$, $\ell_{\max}(S) = \ell_{\max}(R(S))$, $\ell_{\min}(S) = \ell_{\min}(R(S))$, $i_{\max}(S) = i_{\max}(R(S))$, and $i_{\min}(S) = i_{\max}(R(S))$.

Given a *separation constant* $s > 0$, we say that two point sets $A$ and $B$ are *well-separated* if $R(A)$ and $R(B)$ can be enclosed in two balls of radius $r$ such that the distance between the two balls is at least $sr$, where a *ball* of radius $r$ centered at point $c$ is the point set $B = \{p \in \mathbb{R}^d : \text{dist}_2(c, p) \leq r\}$ (see Figure 12.1).

We define the *interaction product* of two point sets $A$ and $B$ as $A \otimes B = \{\{a, b\} : a \in A \wedge b \in B \wedge a \neq b\}$. A *realization* $\mathcal{R}$ of $A \otimes B$ is a set $\{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ with the following properties:

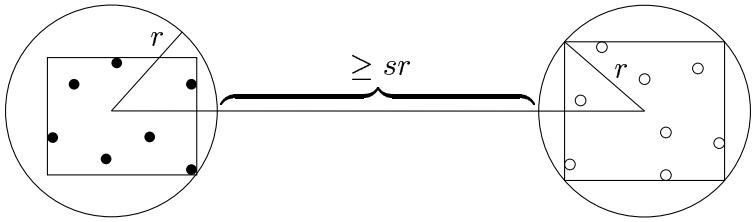



**Figure 12.1**
Two well-separated point sets $A$ (black dots) and $B$ (white dots).

(R1) $A_i \subseteq A$ and $B_i \subseteq B$, for $1 \leq i \leq k$,

(R2) $A_i \cap B_i = \emptyset$, for $1 \leq i \leq k$,

(R3) $(A_i \otimes B_i) \cap (A_j \otimes B_j) = \emptyset$, for $1 \leq i < j \leq k$, and

(R4) $A \otimes B = \bigcup_{i=1}^{k} A_i \otimes B_i$.

Intuitively, this means that for every pair $\{a, b\}$ of distinct points $a \in A$ and $b \in B$, there is a unique pair $\{A_i, B_i\}$ such that $a \in A_i$ and $b \in B_i$. A realization is *well-separated* if it has the following additional property:

(R5) Sets $A_i$ and $B_i$ are well-separated, for $1 \leq i \leq k$.

A binary tree over the points in $S$ defines a recursive partition of $S$ into subsets in a natural manner. In particular, such a tree $T$ has one leaf per point in $S$. An internal node $v$ of $T$ represents the set of points in $S$ corresponding to the leaves of $T$ that are descendants of $v$. We refer to a node representing a subset $A \subseteq S$ as node $A$. A leaf representing point $a \in S$ is referred to as leaf $a$ or node $\{a\}$, depending on the context. We define the *size* of a node $A \in T$ as the cardinality of set $A$. A realization of $A \otimes B$ *uses* a tree $T$ if all sets $A_i$ and $B_i$ in the realization are nodes in $T$. A *well-separated pair decomposition* (WSPD) $\mathcal{D} = (T, \mathcal{R})$ of a point set $S$ consists of a binary tree $T$ over $S$ and a well-separated realization $\mathcal{R}$ of $S \otimes S$ which uses $T$.

Two concepts which are useful when computing well-separated pair decompositions of point sets are those of fair splits and split trees. A *split* of a point set $S$ is

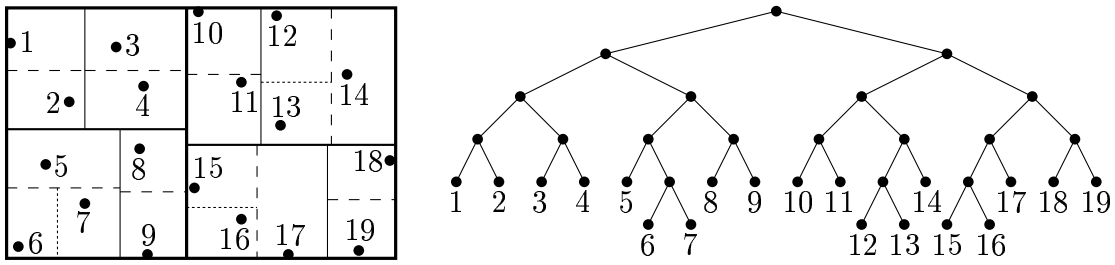



**Figure 12.2**
A partition of a given point set using fair splits and the corresponding fair split tree.

a partition of $S$ into two non-empty point sets lying on either side of a hyperplane perpendicular to one of the coordinate axes and not intersecting any points in $S$. A *split tree* $T$ of $S$ is a binary tree over $S$ defined as follows: If $S = \{x\}$, $T$ contains a single node $\{x\}$. Otherwise, perform a split to partition $S$ into two subsets $S_1$ and $S_2$. Tree $T$ is now constructed from two split trees for point sets $S_1$ and $S_2$ whose roots are the children of the root node $S$ of $T$. For a node $A$ in $T$, the *outer rectangle* $\hat{R}(A)$ is defined as follows: For the root $S$, let $\hat{R}(S)$ be a cube with side length $\ell(\hat{R}(S)) = \ell_{\max}(S)$, centered at the center of $R(S)$. For all other nodes $A$, the hyperplane used for the split of $p(A)$ divides $\hat{R}(p(A))$ into two open boxes. Let $\hat{R}(A)$ be the one that contains $A$. A *fair split* of $A$ is a split of $A$ where the hyperplane splitting $A$ is at distance at least $\ell_{\max}(A)/3$ from each of the two sides of $\hat{R}(A)$ parallel to it. A split tree formed using only fair splits is called a *fair split tree* (Figure 12.2). A *partial fair split tree* $T$ is a subtree of a fair split tree $T'$ containing the root of $T'$. That is, the leaves of $T$ may represent subsets of $S$ instead of single points.

The following are alternative, more restrictive, definitions of outer rectangles and fair splits which in particular ensure that all outer rectangles are boxes. This will be useful in Chapters 13 and 14. The algorithm for constructing a fair split tree makes sure that the splits satisfy these more restrictive conditions. For the root $S$ of $T$, the outer rectangle $\hat{R}(S)$ is defined as above. Given the outer rectangle $\hat{R}(A)$ of a node, a split is fair if it splits $\hat{R}(A)$ perpendicular to its longest side, and the splitting hyperplane has distance at least $\frac{1}{3}\ell_{\max}(\hat{R}(A))$ from the two sides of $\hat{R}(A)$

parallel to it. Let $\tilde{R}(A_1)$ and $\tilde{R}(A_2)$ be the two rectangles produced by this split. We call $\tilde{R}(A_1)$ and $\tilde{R}(A_2)$ the *split rectangles* of $A_1$ and $A_2$, respectively. For $i \in \{1, 2\}$, let $R'_i = \tilde{R}(A_i)$. The following procedure defines $\hat{R}(A_i)$: If $R'_i$ can be split fairly, let $\hat{R}(A_i) = R'_i$. Otherwise, split $R'_i$ perpendicular to its longest side so that the splitting hyperplane has distance at least $\frac{1}{3}\ell_{\max}(R'_i)$ from the two sides of $R'_i$ parallel to it. Only one of the two resulting rectangles is non-empty. Repeat the process, replacing $R'_i$ with this non-empty rectangle.

## 12.2 Searching a Hierarchy of Rectangles

Before describing our algorithm to compute a well-separated pair decomposition of a point set, we describe an algorithm to preprocess an arbitrary binary tree so that certain search queries on the tree can be answered I/O-efficiently. We apply this solution to answer two types of search queries on a hierarchy of nested rectangles represented by a binary tree. These queries arise in Sections 12.3.2 and 13.3.2 as part of our algorithms for constructing a fair split tree and the dumbbell spanner of a point set.

The algorithm combines the ideas of the topology $B$-tree [36] with those of the buffer tree [11] to achieve its goal. Because of this, we call the data structure constructed by the algorithm the *topology buffer tree*, even though it is not an I/O-efficient equivalent of Frederickson's topology tree [77]. In particular, the constructed data structure is static and needs to be rebuilt if the binary tree it represents changes.

In Section 12.2.1, we review the topology tree [77] and characterize the kind of search queries that can be answered efficiently on binary trees using this data structure. In Section 12.2.2, we describe the topology buffer tree, show that it can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os and that a batch of $K$ search queries can be answered in $\mathcal{O}(\text{sort}(N+K))$ I/Os, provided that they meet the conditions defined in Section 12.2.1. Finally, in Section 12.2.3, we apply the topology buffer tree to answer queries on a hierarchy of nested rectangles.

### 12.2.1 The Topology Tree—A Review

The topology tree, introduced by Frederickson [77], is a data structure to represent dynamically changing, possibly unbalanced binary trees so that updates and queries of the tree can be performed in $\mathcal{O}(\log N)$ time. Frederickson proposed this data structure as an alternative implementation of link-cut trees, which were introduced by Sleator and Tarjan [157, 158]. Callahan et al. [36] argue that the topology tree supports INSERT and DELETE operations at the same complexity as LINK and CUT operations. The topology tree $\mathcal{T}$ of a rooted binary tree $T$ is defined as follows:

A *cluster* $C$ in $T$ is the vertex set of a subtree $T'$ of $T$. The *root* of cluster $C$ is the same as the root of tree $T'$. Two disjoint clusters $C_1$ and $C_2$ are adjacent if there exist two vertices $v \in C_1$ and $w \in C_2$ which are adjacent in $T$. A cluster $C_1$ is the *child* of a cluster $C_2$ if the parent of the root $r_1$ of cluster $C_1$ is a node of $C_2$. A *restricted cluster partition* of $T$ is a partition of the vertex set of $T$ into disjoint clusters so that the following conditions hold:

(i) No cluster contains more than two vertices,

(ii) A cluster containing two vertices has at most one child, and

(iii) No two adjacent clusters can be combined without violating Condition (i) or (ii).

Given a binary tree, a *multilevel cluster partition* of a tree $T$ is defined as a sequence $T = T_0, T_1, \ldots, T_r$ of binary trees. Tree $T_i$ is obtained from tree $T_{i-1}$ by contracting every cluster in a restricted cluster partition of $T_{i-1}$ into a single vertex. Tree $T_r$ has a single vertex. Now a *topology* tree $\mathcal{T}$ of tree $T$ is obtained as follows: The vertex set of $\mathcal{T}$ is the disjoint union of the vertex sets of trees $T_0, \ldots, T_r$. The vertices of tree $T_i$ are at level $i$ in $\mathcal{T}$, where level 0 is the level of the leaves of $\mathcal{T}$ and level $r$ is the level of the root of $\mathcal{T}$. A vertex $v \in T_i$ is the parent of a vertex $w \in T_{i-1}$ in $\mathcal{T}$ if the cluster in the cluster partition of $T_{i-1}$ represented by $v$ contains vertex $w$. Since no cluster has size more than two, tree $\mathcal{T}$ is binary.

**Lemma 12.1 (Frederickson [77])** *For all* $1 \leq i \leq r$, $|T_i| \leq \frac{5}{6}|T_{i-1}|$.

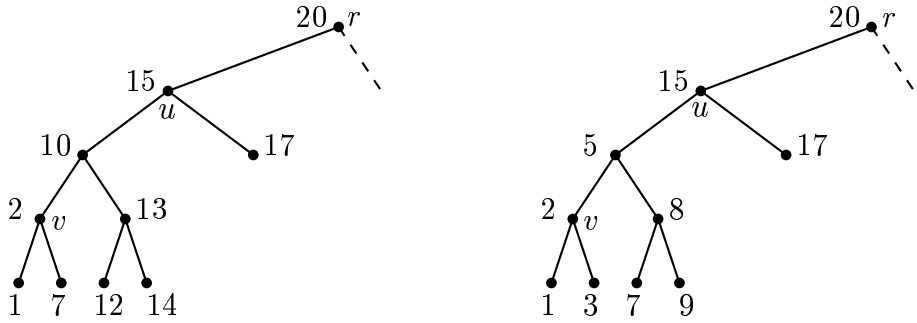



**Figure 12.3**
In the left tree, the answer to the query "Does tree $T$ store element 7?" is stored in tree $T(v)$. In the right tree, the answer is not stored in $T(v)$.

Lemma 12.1 implies that $r = \mathcal{O}(\log N)$ and $|\mathcal{T}| = \mathcal{O}(N)$. Next we characterize the kind of search queries that can be answered efficiently on a binary tree $T$ using its topology tree $\mathcal{T}$. We call query $q$ *oblivious* w.r.t. $T$ if the information stored at every node $v \in T$ is sufficient to decide whether the subtree $T(v)$ of $T$ rooted at $v$ stores an answer to query $q$. (Note that "There is no element stored in tree $T$ matching query $q$" is also a valid answer to query $q$.) That is, if we think about answering query $q$ by traversing a search path in tree $T$ which contains node $v$, then the decision made at vertex $v$ does not depend on decisions made higher up in the tree. In particular, given an ancestor $u$ of $v$ in $T$ so that $T(u)$ contains an answer to query $q$, the information stored at node $v$ is sufficient to decide whether to search $T(v)$ or $T(u) \setminus T(v)$ for an answer to query $q$. A regular binary search query is not oblivious w.r.t. a standard binary search tree: Consider the two search trees shown in Figure 12.3. Then in both cases, the answer to the query "Does tree $T$ store element 7?" is stored in tree $T(u)$. However, the decision whether the answer to this query is stored in $T(v)$ depends on the values stored at $v$ and the parent $p(v)$ of $v$ in $T$. The information stored in the tree can be augmented to make the query oblivious. In particular, every node can be labelled with the possible range of values stored in its subtree, as shown in Figure 12.4.

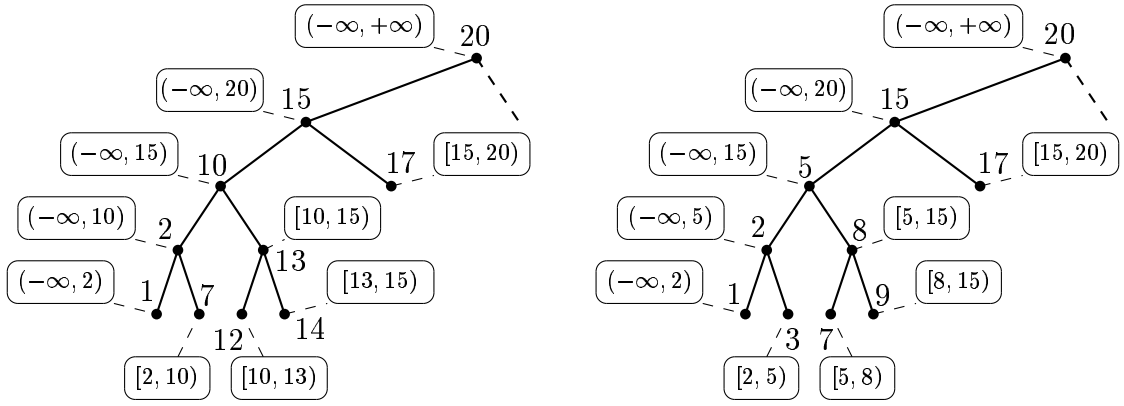



**Figure 12.4**
The trees from Figure 12.3 augmented so that standard binary search queries are oblivious w.r.t. these trees.

The next lemma shows that oblivious queries can be answered using a topology tree.

**Lemma 12.2** *Given a topology tree $\mathcal{T}$ for a binary tree $T$, an oblivious query on $T$ can be answered in $\mathcal{O}(\log N)$ time.*

*Proof.* Consider a node $v \in \mathcal{T}$. We define the set $\mathrm{Desc}_0(v)$ as the set of vertices in $T_0$ which are descendants of $v$ in $\mathcal{T}$. Let $T_v$ be the subtree of $T$ induced by the vertices in $\mathrm{Desc}_0(v)$, and let $r_v$ be the root of $T_v$. Let every node $v \in \mathcal{T}$ be augmented with the information stored at node $r_v \in T$.

Now let $v \in T_i$ and assume that an answer to query $q$ is stored in tree $T_v$. This is true for the root $r$ of $\mathcal{T}$, as $T_r = T$. If $v$ has one child $w$ in $\mathcal{T}$, then $T_v = T_w$, and obviously an answer to query $q$ is stored in tree $T_w$, so that the search proceeds to node $w$. Otherwise, let $u$ and $w$ be the two children of $v$. As $u$ and $w$ are in the cluster represented by $v$, they are adjacent. W.l.o.g., let $u$ be the parent of $w$. As query $q$ is oblivious, the information stored with $w$ is sufficient to decide whether the subtree $T(r_w)$ rooted at $r_w$ stores an answer to query $q$. If this is the case, the search continues to node $w$. Otherwise, it proceeds to $u$. This procedure takes $\mathcal{O}(\log N)$ time, as it visits only one node per level of $\mathcal{T}$. We have to show that it

is correct. To do this, we show that this search maintains the following invariant: *If the search visits node $v$, then the nodes in $T(r_v) \setminus T_v$ do not store an answer to query $q$.* The invariant immediately implies the correctness of the search procedure: If tree $T(r_w)$ does not contain an answer to query $q$, tree $T_w$ cannot contain such an answer, so that an answer to query $q$ must be stored in tree $T_u$. Otherwise, the invariant implies that no node in $T(r_w) \setminus T_w$ stores an answer to query $q$ because the search visits node $w$. Hence, the answer to query $q$ stored in $T(r_w)$ must be stored in $T_w$. We have to show the invariant.

The invariant holds for the root $r$ of $\mathcal{T}$, as $T_r = T$, and $T(r) \setminus T_r = \emptyset$. Now assume that the invariant holds for a node $v$. If $v$ has only one child $w$, the invariant holds for $w$, as $T_w = T_v$ and $T(r_v) = T(r_w)$. Otherwise, let $u$ and $w$ be the children of $v$ as defined above. Then $T(r_w) \subseteq T(r_v)$ and $T_w = T(r_w) \cap T_v$, so that no node in $T(r_w) \setminus T_v = T(r_w) \setminus T_w$ stores an answer to query $q$. Hence, if the search visits node $w$, the invariant is maintained. On the other hand, $T(r_u) = T(r_v)$ and $T(r_u) \setminus T_u = (T(r_v) \setminus T_v) \cup T_w = (T(r_v) \setminus T_v) \cup T(r_w)$. If the search visits node $u$, tree $T(r_w)$ does not contain an answer to query $q$ and no node in $T(r_v) \setminus T_v$ stores an answer to query $q$. Hence, no node in $T(r_u) \setminus T_u$ stores an answer to query $q$. □

### 12.2.2 The Topology Buffer Tree

By Lemma 12.2, the topology tree is a useful tool to answer oblivious search queries on a binary tree $T$ efficiently. Next we show how to derive a modified version of the topology tree which can be used to answer a batch of oblivious search queries I/O-efficiently. Intuitively, we build a topology tree $\mathcal{T}$ for $T$ and cut it into layers of height $\frac{1}{2}\log(M/B)$. The topology buffer tree $\mathcal{B}$ of $T$ is obtained by contracting every subtree in such a layer into a single node. Tree $\mathcal{B}$ can be used to simulate searching $\mathcal{T}$: Filter all queries in the given set $Q$ of queries through $\mathcal{B}$, from the root toward the leaves. For every node $v \in \mathcal{B}$, load the subtree $\mathcal{T}_v$ of $\mathcal{T}$ represented by $v$ into internal memory and filter the queries in the set $Q_v$ of queries assigned to $v$ through $\mathcal{T}_v$ to determine for every query, the child of $v$ to which this query is assigned next. Since

tree $\mathcal{T}_v$ is guaranteed to have size at most $2\sqrt{M/B}$ and no more than $\sqrt{M/B}$ children, we can use the distribution algorithm of [138] to implement the filtering of queries through tree $\mathcal{B}$. However, in order to bound the height of $\mathcal{B}$ by $\mathcal{O}(\log_{M/B}(N/B))$ and the overhead I/Os incurred by the algorithm of [138] when distributing the queries of one node to its children, we would like to guarantee that tree $\mathcal{B}$ has size $\mathcal{O}(N/M)$. This is not easily achieved once tree $\mathcal{T}$ has been constructed, so that we choose a different approach, which builds a two-level data structure. The first level of the data structure is used to identify a small subtree of $T$ containing the answer to query $q$. The second level of the data structure represents these small subtrees and can be used to compute the final answer to each query $q \in Q$.

First we develop a modified procedure for answering oblivious queries on $T$ using a topology tree for a compressed version of $T$ and then finishing the search in a small subtree of $T$. Given a binary tree $T$, we define a compressed tree $\tilde{T}$ as follows: First we reduce the number of leaves of $T$ to $\mathcal{O}(N/M)$. Let $L$ be the set of nodes in $T$ so that for every node $v \in L$, $|T(v)| \leq M$, and $|T(p(v))| > M$. Observe that no node in $L$ has another node in $L$ as an ancestor. For every node $v \in L$, let $T_v = T(v)$. We remove all proper descendants of $v$ from $T$. Let $T'$ be the resulting tree. The leaves of $T'$ are the nodes in $L$. Now let $P_1, \ldots, P_k$ be the maximal paths in $T'$ whose nodes have exactly one child in $T'$. We partition each such path $P_i$ into a minimum number of subpaths $P_{i,1}, \ldots, P_{i,q}$ so that no path $P_{i,j}$ has size more than $M$. For a path $P_{i,j}$, let $v$ be the topmost node of $P_{i,j}$ and $w$ be the bottommost node of $P_{i,j}$. Then we define $T_v = P_{i,j}$, make the child of $w$ the child of $v$, and remove all nodes in $P_{i,j} - \{v\}$ from $T'$. Let $\tilde{T}$ be the tree obtained after performing this operation for all paths $P_{i,j}$. Finally let $T_v = (\{v\}, \emptyset)$, for all internal nodes of $\tilde{T}$ with two children.

**Lemma 12.3** *Tree $\tilde{T}$ has size $\mathcal{O}(N/M)$.*

*Proof.* The number of leaves in $\tilde{T}$ is the same as the number of leaves in $T'$. The number of internal nodes with two children as well as the number of paths $P_1, \ldots, P_k$ is at most twice the number of leaves in tree $T'$. The total number of internal nodes with one child in $\tilde{T}$ is bounded by the number of paths $P_{i,j}$ into which paths $P_1, \ldots, P_k$ are

partitioned. This number can be bounded by $\sum_{i=1}^{k} \lceil |P_i|/M \rceil \leq k + \left(\sum_{i=1}^{k} |P_i|\right)/M \leq k + N/M$. Since $k \leq 2|L|$, the total number of nodes in $\tilde{T}$ is $\mathcal{O}(|L|) + N/M$. To bound the number of leaves, we observe that for each such leaf $v$, the tree $T(p(v))$ in $T$ has size more than $M$. Hence, there are less than $N/M$ such parents in $T$. Since every node in $T$ has at most two children, the number of leaves is thus bounded by $2N/M$, and the size of $\tilde{T}$ is $\mathcal{O}(N/M)$. □

Now let $\tilde{\mathcal{T}}$ be a topology tree for $\tilde{T}$. Then trees $\tilde{\mathcal{T}}$ and $T_v$, $v \in \tilde{T}$, can be used to answer an oblivious search query on $T$. To answer such a query $q$, we search tree $\tilde{\mathcal{T}}$ to identify the leaf $v \in \tilde{\mathcal{T}}$ so that tree $T_v$ stores the answer to query $q$. This can be done using Lemma 12.2. Given leaf $v$, we search tree $T_v$ to find the answer to query $q$.

A topology buffer tree $\mathcal{B} = \left(\tilde{\mathcal{B}}, (T_v)_{v \in \tilde{T}}\right)$ for $T$ consists of two parts: A list of trees $T_v$, $v \in \tilde{T}$, and a tree $\tilde{\mathcal{B}}$ constructed from $\tilde{\mathcal{T}}$ as follows: We cut $\tilde{\mathcal{T}}$ into layers so that the $i$-th layer contains the subtrees of $\tilde{\mathcal{T}}$ induced by the vertices on levels $ih, \ldots, (i+1)h$, where $h = \frac{1}{2}\log(M/B)$ is the height of a layer. Note that two successive layers overlap. Let $\mathcal{T}_1, \ldots, \mathcal{T}_q$ be the subtrees of all layers. Tree $\tilde{\mathcal{B}}$ contains one node $v_i$ per tree $\mathcal{T}_i$. For every node $v_i$, we define $\mathcal{T}_{v_i} = \mathcal{T}_i$. Node $v_i$ is a child of another node $v_j$ if the root of tree $\mathcal{T}_i$ is a leaf of $\mathcal{T}_j$.

**Observation 12.1** *Tree $\tilde{\mathcal{B}}$ has height $\mathcal{O}(\log_{M/B}(N/B))$ and $\mathcal{O}(N/M)$ nodes. Every node $v \in \tilde{\mathcal{B}}$ has degree $\mathcal{O}\left(\sqrt{M/B}\right)$. Tree $\mathcal{T}_v$ has size $\mathcal{O}\left(\sqrt{M/B}\right)$, for all $v \in \tilde{\mathcal{B}}$.*

To use the topology buffer tree $\mathcal{B}$ for answering oblivious search queries on $T$, we represent it on disk as follows: We number the nodes of $\mathcal{B}$ level by level, from left to right. Then we store their corresponding subtrees $\mathcal{T}_1, \ldots, \mathcal{T}_q$ of $\tilde{\mathcal{T}}$ sorted in this order, striped across all $D$ disks. Trees $T_v$, $v \in \tilde{T}$, are stored striped across all $D$ disks, sorted according to some total order defined on the nodes of $\tilde{T}$. The next lemma shows that such a representation of $\mathcal{B}$ can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os. Theorem 12.1 shows that $\mathcal{B}$ can be used to answer a batch of $K$ oblivious search queries on $T$ in $\mathcal{O}(\text{sort}(N+K))$ I/Os.

**Procedure** BUILDTOPOLOGYBUFFERTREE

**Input:** A binary tree $T$ with $N$ nodes.
**Output:** A topology buffer tree $\mathcal{B} = (\tilde{\mathcal{B}}, (T_v)_{v \in \tilde{\mathcal{T}}})$ of $T$.

**Part 1—Construct tree $\tilde{T}$ and trees $T_v$, $v \in \tilde{T}$:**

1: For all nodes $v \in T$, compute the size $|T(v)|$ of the subtree $T(v)$ of $T$ rooted at $v$.
2: For all nodes $v \in T$ with $|T(v)| \leq M < |T(p(v))|$, mark all descendants of $v$ as belonging to $T_v = T(v)$. Append them to a list $X$ representing trees $T_v$, and remove all proper descendants of $v$ from $T$.
3: Process the resulting tree $T'$ from the root toward the leaves:
   At every node $v$ which has one child and whose parent does not exist or has two children, start a new path $P_{i,j}$ and send the identity of $v$ and the size of path $P_{i,j}$ to the child of $v$.
   At every node $v$ which has one child and whose parent has one child, let $P_{i,j}$ be the path containing $p(v)$, and let $u$ be the first node on this path. If $|P_{i,j}| < M$, increase $|P_{i,j}|$ by one and pass $u$ and $|P_{i,j}|$ to the child of $v$, thereby adding $v$ to $P_{i,j}$. If $|P_{i,j}| = M$, start a new path at $v$ and set $p(v) = u$.
   At every node $v$ which has zero or two children and whose parent has one child, let $u$ be the first vertex on the path $P_{i,j}$ containing $p(v)$. Then set $p(v) = u$.
4: Scan the vertex set of $T'$ to append all internal nodes of $T'$ to list $X$, labelled with the paths containing them. (Every node with two children is assumed to form a path of its own.)
5: Process tree $T'$ from the leaves toward the root and update the pointers of all nodes in $\tilde{T}$ to their children in $\tilde{T}$. Discard all nodes whose parent is contained in the same path $P_{i,j}$.
6: Sort the nodes in list $X$ by the labels of the trees $T_v$ containing them.

---

**Algorithm 12.1**
Building a topology buffer tree.

**Lemma 12.4** *A topology buffer tree $\mathcal{B}$ representing a binary tree $T$ of size $N$ can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space.*

*Proof.* Procedure BUILDTOPOLOGYBUFFERTREE presented in Algorithms 12.1 and 12.2 constructs $\mathcal{B}$. It is easy to verify that this procedure is correct. We analyze its I/O-complexity.

Step 1 of the first part of the algorithm can be realized by processing the nodes of $T$ from the leaves toward the root using time-forward processing. Step 2 applies

**Procedure** BUILDTOPOLOGYBUFFERTREE (CONTINUED)

**Part 2—Construct $\tilde{\mathcal{T}}$:**

1: $T_0 \leftarrow \tilde{T}$
2: $i \leftarrow 0$
3: **while** $|T_i| > 1$ **do**
4: Process tree $T_i$ from the leaves toward the root, making a node of $T_i$ form a cluster with one of its children if the resulting cluster satisfies the conditions of a cluster partition and this child is not part of a cluster yet.
5: Replace every cluster of $T_i$ with a single vertex $v$, using procedure CONTRACTGRAPH, and label every such vertex $v$ with the at most two nodes in its cluster, thereby making them children of $v$ in $\tilde{\mathcal{T}}$. Call the resulting graph $T_{i+1}$.
6: $i \leftarrow i + 1$
7: **end while**

**Part 3—Construct $\tilde{\mathcal{B}}$:**

1: Apply the Euler tour technique and list ranking to compute a preorder numbering $\nu$ of $\tilde{\mathcal{T}}$ and label every node $v$ of $\tilde{\mathcal{T}}$ with its distance $d(v)$ from the root.
2: Scan the vertex set of $\tilde{\mathcal{T}}$ to label every node $\tilde{\mathcal{T}}$ with the distance $d'(v) = \lfloor d(v)/h \rfloor$. For every node $v$ so that $d(v)$ is a multiple of $h$, create an additional copy of $v$ whose label is $d'(v) = \lfloor d(v)/h \rfloor + 1$.
3: Sort the nodes of $\tilde{\mathcal{T}}$ lexicographically by their label pairs $(d'(v), \nu(v))$.

---

**Algorithm 12.2**
Algorithm 12.1 continued.

time-forward processing to process the nodes of $T$ from the root toward the leaves. Steps 3 and 5 apply time-forward processing to tree $T'$. Step 4 scans the vertex set of $T'$. Finally, Step 6 sorts the vertex list $X$, which has size $N$. Hence, the first part of the algorithm takes $\mathcal{O}(\text{sort}(N))$ I/Os.

Every iteration of the loop in Lines 3–7 of the second part of the algorithm takes $\mathcal{O}(\text{sort}(|T_i|))$ I/Os, using time-forward processing to realize the computation of Line 4 and procedure CONTRACTGRAPH to realize the computation of Line 5. By Lemma 12.1, the sizes of trees $T_0, T_1, \ldots$ are geometrically decreasing, so that Part 2 takes $\mathcal{O}(\text{sort}(N))$ I/Os.

Line 1 of the third part of the algorithm takes $\mathcal{O}(\text{sort}(N/M))$ I/Os [43]. Lines 2 and 3 take $\mathcal{O}(\text{sort}(N/M))$ I/Os, since they scan and sort the vertex set of $\tilde{\mathcal{T}}$ once, and this set has size $\mathcal{O}(N/M)$. Hence, the third part takes $\mathcal{O}(\text{sort}(N/M))$ I/Os, which proves the lemma. □

Given a topology buffer tree $\mathcal{B}$ for a binary tree $T$, we use Algorithm 12.3 to answer a batch $Q$ of $K$ oblivious search queries on $T$. In this algorithm, we denote the set of queries to pass through a node $v \in \tilde{\mathcal{B}}$ by $Q_v$. Next we show that Algorithm 12.3 takes $\mathcal{O}(\text{sort}(N+K))$ I/Os.

**Theorem 12.1** *A batch of $K$ oblivious search queries on a binary tree $T$ of size $N$ can be answered in $\mathcal{O}(\text{sort}(N+K))$ I/Os using $\mathcal{O}((N+K)/B)$ blocks of external memory.*

*Proof.* The correctness of Algorithm 12.3 follows from Lemma 12.2. In particular, this lemma implies that the first part of the algorithm (Lines 1–12) identifies the correct node $v$ of $\tilde{T}$ for every query $q$ so that tree $T_v$ stores the answer to query $q$. Then the second part of the algorithm (Lines 13–18) finishes answering query $q$ by searching tree $T_v$.

In order to apply Algorithm 12.3, the topology buffer tree $\mathcal{B}$ has to be constructed. By Lemma 12.4, this takes $\mathcal{O}(\text{sort}(N))$ I/Os. Next we analyze the I/O-complexity of Algorithm 12.3.

Reading subtrees $\mathcal{T}_v$ and $T_v$ into internal memory in Lines 7 and 15 of the algorithm takes $\mathcal{O}(\text{scan}(N))$ I/Os in total, as the total size of these trees is $\mathcal{O}(N)$, and these trees are stored striped across all $D$ disks. Filtering the queries through trees $\mathcal{T}_v$ and $T_v$ in Lines 9 and 17 does not incur any I/Os except for reading the queries and writing the results because trees $\mathcal{T}_v$ and $T_v$ fit into main memory. Since every node in $\tilde{\mathcal{B}}$ has $\mathcal{O}\left(\sqrt{M/B}\right)$ children, the distribution procedure of [138] performs $\mathcal{O}(\text{scan}(|Q_v|)+k)$ I/Os per node $v \in \tilde{\mathcal{B}}$ to distribute the queries in $Q_v$ to its children $w_1, \ldots, w_k$ in a manner that allows every set $Q_{w_i}$ to be read in $\mathcal{O}(\text{scan}(|Q_{w_i}|))$ I/Os, once the first block on each disk is known. The $\mathcal{O}(k)$ overhead I/Os are caused by

**Procedure** BATCHEDOBLIVIOUSSEARCH

**Input:** A topology buffer tree $\mathcal{B} = (\tilde{\mathcal{B}}, (T_v)_{v \in \tilde{T}})$ representing a binary tree $T$ and a batch $Q = \{q_1, \ldots, q_K\}$ of oblivious search queries on $T$.

**Output:** The set $A = \{a_1, \ldots, a_K\}$ of answers to the queries in $Q$.

1: Scan list $Q$ to augment every disk block storing queries from $Q$ with a pointer to the next such block on the same disk.
2: Let $S_1$ be the list of $D$ pointers to the first block of queries on each disk.
3: Let $h$ be the height of tree $\mathcal{B}$.
4: **for** $i = 1, \ldots, h$ **do**
5: $S_{i+1} \leftarrow \emptyset$
6: **for** every node $v$ at level $i$, from left to right **do**
7: Load tree $\mathcal{T}_v$ into internal memory.
8: Load the next $D$ items in $S_i$ into internal memory to find the start of set $Q_v$ on each disk.
9: Filter the queries in $Q_v$ through $\mathcal{T}_v$ and use the distribution algorithm of [138] to create sets $Q_{w_1}, \ldots, Q_{w_k}$, where $w_1, \ldots, w_k$ are the children of $v$ in $\tilde{\mathcal{B}}$, sorted from left to right.
10: The distribution algorithm generates a list $S_{w_i}$ of $D$ pointers to the first block on each disk storing queries in $Q_{w_i}$. Append lists $S_{w_1}, \ldots, S_{w_k}$ to $S_{i+1}$, in this order.
11: **end for**
12: **end for**
13: Reorder the sublists $S_v$, $v \in \tilde{T}$, of $S_{h+1}$ in the same order as trees $T_v$.
14: **for** every node $v \in \tilde{T}$ **do**
15: Load tree $T_v$ into internal memory.
16: Load the next $D$ items from $S_{h+1}$ into internal memory to find the start of set $Q_v$ on each disk.
17: Filter the queries in $Q_v$ through $T_v$ and write the answer to every query to disk.
18: **end for**

**Algorithm 12.3**
Answering oblivious search queries using a topology buffer tree.

reading set $S_v$ and writing sets $S_{w_1}, \ldots, S_{w_k}$. Also, the last I/O spent on writing sets $Q_{w_1}, \ldots, Q_{w_k}$ may write less than $DB$ queries, so that this I/O cannot be charged to a sufficient number of queries in $Q_v$. The $\mathcal{O}(k)$ additional I/Os can be charged to children $w_1, \ldots, w_k$, thereby charging no node of $\tilde{\mathcal{B}}$ for more than $\mathcal{O}(1)$ overhead I/Os. The overhead I/Os for generating list $S_{h+1}$ can be charged to the nodes of $\tilde{T}$, again charging a node in $\tilde{T}$ for $\mathcal{O}(1)$ I/Os. By Lemma 12.3 and Observation 12.1, trees $\tilde{\mathcal{B}}$ and $\tilde{T}$ have size $\mathcal{O}(N/M)$, so that the total number of overhead I/Os can be bounded by $\mathcal{O}(N/M) = \mathcal{O}(\text{scan}(N))$. The total number of non-overhead I/Os performed in Phase 1 is $\mathcal{O}\left(\text{scan}\left(\sum_{v \in \tilde{\mathcal{B}}} |Q_v|\right)\right)$. However, $\sum_{v \in \tilde{\mathcal{B}}} |Q_v| = \mathcal{O}\left(K \log_{\frac{M}{B}} \frac{N}{M}\right)$ because every query is contained in $\mathcal{O}\left(\log_{\frac{M}{B}} \frac{N}{M}\right)$ sets $Q_v$. Hence, the first part of the algorithm takes $\mathcal{O}(\text{sort}(N+K))$ I/Os.

The second part of the algorithm takes $\mathcal{O}(\text{scan}(N+K))$ I/Os because reading subtrees $T_v$ takes $\mathcal{O}(\text{scan}(N))$ I/Os, as argued above, all query sets can be read in $\mathcal{O}(\text{scan}(K) + |\tilde{T}|) = \mathcal{O}(\text{scan}(N) + \text{scan}(K))$ I/Os, and writing all answers to queries $q_1, \ldots, q_K$ to disk takes $\mathcal{O}(\text{scan}(K))$ I/Os, as they are written in the order in which the queries are read, which can be done in a striped manner.

Summing the complexities of constructing tree $\mathcal{B}$ and the two parts of Algorithm 12.3, we obtain the theorem. □

### 12.2.3 Querying a Hierarchy of Rectangles

Next we show how to use the topology buffer tree introduced in Section 12.2.2 to answer two types of queries on a tree $T$ and a point set $S$ which arise as part of the algorithms for constructing a fair split tree and the dumbbell spanner of a point set in Sections 12.3.2 and 13.3.2. For the first type of query, tree $T$ represents a hierarchy of nested rectangles. For the second type of query, $T$ is a fair split tree, hence representing a hierarchy of nested rectangles with additional properties.

#### 12.2.3.1 Deepest Containment Queries

Let $S \subset \mathbb{R}^d$ be a point set and $T$ be a tree with the following three properties:

(i) Every node $v \in T$ has an associated rectangle $R(v)$,

(ii) For the root $r$ of $T$, $R(r)$ contains all points in $S$,

(iii) For every node $v \neq r$ in $T$, $R(v) \subseteq R(p(v))$, and

(iv) If $R(v) \cap R(w) \neq \emptyset$, then w.l.o.g. $R(v) \subseteq R(w)$ and $w$ is an ancestor of $v$.

A *deepest containment query* on $T$ is the following: *Given a point $p \in S$, find the node $v \in T$ so that $p \in R(v)$ and there is no descendant of $v$ satisfying this condition.* We show how to answer these queries for all points in $S$ I/O-efficiently.

**Lemma 12.5** *Given a set $S \subset \mathbb{R}^d$ of $N$ points and a tree $T$ with Properties (i)–(iv) above, deepest containment queries on $T$ for all points $p \in S$ can be answered in $\mathcal{O}(\mathrm{sort}(N + |T|))$ I/Os and linear space.*

*Proof.* We can apply Theorem 12.1 to solve this problem in $\mathcal{O}(\mathrm{sort}(N + |T|))$ I/Os and linear space if we can show that a deepest containment query is oblivious w.r.t. tree $T$. To see the latter, let $p \in S$ be a query point and $v$ be the answer to a deepest containment query on $T$ with point $p$. Then $p \in R(w)$ if and only if $w$ is a node on the path from $v$ to $r$ in $T$. Hence, tree $T(w)$ contains the answer to the query if and only if $p \in R(w)$. The latter condition can be tested by comparing the coordinates of $p$ with the coordinates of the hyperplanes defining rectangle $R(w)$. Hence, deepest containment queries are oblivious w.r.t. $T$. □

#### 12.2.3.2 Restricted Containment Queries

Let $T$ be a fair split tree of some point set $S \subset \mathbb{R}^d$. A *restricted containment query* on $T$ is the following: *Given a point $p \in \mathbb{R}^d$ and two real numbers $x$ and $y$, report all nodes $v \in T$ so that $p \in \tilde{R}(A)$, $\ell_{\max}(A) \leq x$ and $\ell_{\min}(\tilde{R}(A)) \geq y$.*

To develop an I/O-efficient solution to this problem, we need to show a few technical lemmas. For a node $A \in T$, we denote its *$k$-th ancestor* by $p^{(k)}(A)$. In particular, $p^{(0)}(A) = A$, $p^{(i+1)}(A) = p(p^{(i)}(A))$ if $p^{(i)}(A)$ is not the root of $T$, and $p^{(i+1)}(A) = p^{(i)}(A)$ if $p^{(i)}(A)$ is the root of $T$.

**Lemma 12.6** *Let $A$ be a node in a fair split tree $T$ whose distance to the root of $T$ is at least $d$, and let $A' = p^{(d)}(A)$. Then $\ell_{\max}(\hat{R}(A)) \leq 3\ell_{\max}(A')$.*

*Proof.* Let $A' = A_0, A_1, \ldots, A_d = A$ be the nodes on the path from $A'$ to $A$ in $T$. For $j = 0, \ldots, d$, let $i_j$ be the dimension so that rectangle $\hat{R}(A_j)$ is split using a hyperplane $H_j$ perpendicular to dimension $i_j$. Because there are $d+1$ dimensions $i_0, \ldots, i_d$, there have to be two indices $0 \leq x < y \leq d$, so that $i_x = i_y$. Since $\ell_{\max}(A_x) \leq \ell_{\max}(A')$, and at least one point of $A_x$ has to lie on either side of hyperplane $H_x$, no point in $A_{x+1}$ has distance more than $\ell_{\max}(A')$ from hyperplane $H_x$. Since $A_y \subseteq A_{x+1}$, and at least one point of $A_y$ has to lie on either side of hyperplane $H_y$, hyperplane $H_y$ has distance at most $\ell_{\max}(A')$ from $H_x$. Rectangle $\hat{R}(A_y)$ lies completely to one side of $H_x$, namely on the same side as hyperplane $H_y$. On the other hand, at least one third of $\hat{R}(A_y)$ has to lie on either side of $H_y$ because the split is fair. Hence, at least one third of $\hat{R}(A_y)$ lies between $H_x$ and $H_y$, so that $\ell_{i_y}(\hat{R}(A_y)) \leq 3\ell_{\max}(A')$. Since the outer rectangles of nodes in the split tree are always split perpendicular to their longest side, $\ell_{\max}(\hat{R}(A_y)) = \ell_{i_y}(\hat{R}(A_y))$. In order to see that $\ell_{\max}(\hat{R}(A)) \leq 3\ell_{\max}(R(A'))$, it remains to observe that $\hat{R}(A) \subseteq \hat{R}(A_y)$, so that $\ell_{\max}(\hat{R}(A)) \leq \ell_{\max}(\hat{R}(A_y))$. □

**Lemma 12.7** *Let $A$ be a node in a fair split tree $T$ whose distance from the root of $T$ is at least $d$, and let $A' = p^{(d)}(A)$. Then $\ell_{\max}(\hat{R}(A)) \leq \frac{2}{3}\ell_{\max}(\hat{R}(A'))$.*

*Proof.* We distinguish two cases. If every dimension is split exactly once along the path from $A'$ to $A$, then $\ell_{\max}(\hat{R}(A)) \leq \frac{2}{3}\ell_{\max}(\hat{R}(A'))$ because $\ell_i(\hat{R}(A')) \leq \ell_{\max}(\hat{R}(A'))$, for $1 \leq i \leq d$, and every dimension is split fairly. If there is a dimension which is not split along this path, then there has to be a dimension $i$ which is split twice. Let $A''$ be the node where the second split in dimension $i$ occurs. As dimension $i$ has been split before, $\ell_i(\hat{R}(A'')) \leq \frac{2}{3}\ell_{\max}(\hat{R}(A'))$. On the other hand, every outer box is split perpendicular to its longest dimension, so that $\ell_{\max}(\hat{R}(A'')) = \ell_i(\hat{R}(A''))$. The lemma now follows because $\hat{R}(A) \subseteq \hat{R}(A'')$ and hence $\ell_{\max}(\hat{R}(A)) \leq \ell_{\max}(\hat{R}(A''))$. □

Using Lemmas 12.6 and 12.7, we can bound the number of nodes in $T$ reported for a restricted containment query.

**Lemma 12.8** *Let $T$ be a fair split tree, let $p$ be a point in $\mathbb{R}^d$, and let $x$ and $y$ be two real numbers. Then there are $\mathcal{O}(d \log(x/y))$ nodes in $T$ matching a restricted containment query for point $p$ with parameters $x$ and $y$.*

*Proof.* Assume w.l.o.g. that $p \in \hat{R}(S)$, where $S$ is the root of tree $T$ because otherwise no node of $T$ matches the query. Then all nodes matching the query appear along a root-to-leaf path in $T$ because the nodes at any level in $T$ are disjoint. Let $A$ be a node in $T$ which matches the query and so that no ancestor of $A$ matches the query. Let $B$ be a node in $T$ which matches the query and so that no descendant of $B$ matches the query. It suffices to show that $A = p^{(k)}(B)$, for some $k = \mathcal{O}(d \log(x/y))$. Assume w.l.o.g. that $k > d$ because otherwise, the lemma holds.

Since node $A$ matches the query, $\ell_{\max}(A) \leq x$. Let $A'$ be the node on the path from $A$ to $B$ so that $A = p^{(d)}(A')$. Since $k > d$, node $A'$ exists. By Lemma 12.6, $\ell_{\max}(\hat{R}(A')) \leq 3\ell_{\max}(A)$.

Since node $B$ matches the query, $\ell_{\min}(\tilde{R}(B)) \geq y$ and hence $\ell_{\max}(\hat{R}(p(B)) \geq y$. Since $k > d$, $A'$ is an ancestor of $p(B)$. Since $\ell_{\max}(\hat{R}(A')) \leq 3x$ and $\ell_{\max}(\hat{R}(p(B)) \geq y$, it follows from Lemma 12.7 that $A' = p^{(l)}(p(B))$, where $l \leq d \log_{3/2}(3x/y)$. Hence, $k \leq 1 + d(1 + \log_{3/2}(3x/y)) = \mathcal{O}(d \log(x/y))$. □

We are now ready to present our algorithm for answering a batch $Q$ of restricted containment queries on a fair split tree $T$. We use a topology buffer tree $\mathcal{B}$ representing tree $T$ to answer the queries in $Q$. However, we cannot apply Theorem 12.1 directly because the algorithm is required to report *all* nodes of $T$ satisfying the query. We modify the search algorithm of Lemma 12.2 so that it reports all nodes matching a query $q$, given a topology tree $\mathcal{T}$ representing $T$. Below we argue that this procedure leads to an I/O-efficient algorithm, using a topology buffer tree $\mathcal{B}$ of $T$ instead of tree $\mathcal{T}$.

So let $\mathcal{T}$ be a topology tree representing the fair split tree $T$, and let $q = (p_q, x_q, y_q)$ be a query with point $p_q \in \mathbb{R}^d$ and parameters $x_q$ and $y_q$. Then we filter query $q$ through tree $\mathcal{T}$, starting at the root. For every node $v \in \mathcal{T}$ reached by query $q$ during the query procedure, we apply the following rules: If $v$ is a leaf and node $r_v$ matches the query, we report $r_v$ as the pair $(q, r_v)$. If $v$ is an internal node with one child $w$, we forward query $q$ to node $w$. If $v$ is an internal node with two children $u$ and $w$, assume w.l.o.g. that tree $T_u$ contains the parent of the root $r_w$ of tree $T_w$. In this case, we call $u$ the *master* and $w$ the *slave* of node $v$. If query point $p_q$ is not contained in $\tilde{R}(r_w)$, we send query $q$ to node $u$. Otherwise, we distinguish three cases: If $\ell_{\min}(\tilde{R}(r_w)) < y_q$, we continue the search at $u$ only. If $\ell_{\max}(r_w) > x_q$, we continue the search at $w$ only. Otherwise, we make another copy of query $q$ and send one copy to each of the two children $u$ and $w$ of $v$. The following lemma shows that this procedure is correct and I/O-efficient.

**Lemma 12.9** *Given a fair split tree $T$ of a set $S$ of $N$ points in $\mathbb{R}^d$ and a set $Q$ of $K$ restricted containment queries, the queries in set $Q$ can be answered on $T$ in $\mathcal{O}(\mathrm{sort}(N+K+P))$ I/Os using $\mathcal{O}((N+K+P)/B)$ blocks of external memory, where $P = \mathcal{O}\left(d \sum_{q \in Q} \log x_q/y_q\right)$ is the number of nodes reported for all queries in $Q$.*

*Proof.* In order to make the above search procedure I/O-efficient, we use a topology buffer tree $\mathcal{B}$ instead of the topology tree $\mathcal{T}$ representing $T$. We filter the queries in $Q$ through $\mathcal{B}$ using Algorithm 12.3. Once all queries have been processed, we sort the reported pairs $(q, r_v)$ lexicographically, thereby storing all answers to query $q$ consecutively. By Theorem 12.1, the I/O-complexity of this solution is bounded by $\mathcal{O}\left(\mathrm{sort}\left(N + \sum_{q \in Q} \alpha(q)\right)\right)$, where $\alpha(q)$ is the number of copies of query $q$ created by the procedure. Below we show that $\alpha(q) = 1 + \mathcal{O}(P_q)$, where $P_q$ is the number of nodes reported for query $q$. By Lemma 12.8, $P_q = \mathcal{O}(d \log(x_q/y_q))$, so that the I/O and space bounds as claimed in the lemma follow. Before showing that $\alpha(q) = 1 + \mathcal{O}(P_q)$, for all $q \in Q$, we prove that the procedure above reports all nodes matching query $q$.

It is obvious that the procedure deals with leaves of $\mathcal{T}$ and nodes with one child correctly, as there are no choices to be made. So let $v$ be a node with master $u$

and slave $w$. If the search does not continue to $w$, there are two possibilities: either $p_q \notin \tilde{R}(r_w)$, or $\ell_{\min}(\tilde{R}(r_w)) < y_q$. Since $\tilde{R}(A) \subseteq \tilde{R}(r_w)$, for all nodes $A \in T_w$, no node in $T_w$ matches query $q$ in either of the two cases. If the search does not continue to $u$, $p_q \in \tilde{R}(r_w)$ and $\ell_{\max}(r_w) > x_q$. Since $p_q \in \tilde{R}(r_w)$, every node $A \in T_u$ matching the query is an ancestor of $r_w$, so that $\ell_{\max}(A) \geq \ell_{\max}(r_w) > x_q$. Hence, no node in $T_u$ matches the query in this case. We have shown that the procedure visits a node $v \in \mathcal{T}$ unless it has proof that no node in $T_v$ is an answer to the query. Hence, it is guaranteed to find all nodes matching the query.

It remains to show that $\alpha(q) = 1 + \mathcal{O}(P_q)$, for every query $q \in Q$. To see this, observe that when query $q$ is duplicated at a node $v$ with slave $w$, node $r_w$ matches the query because in this case, $p_q \in \tilde{R}(r_w)$, $\ell_{\max}(r_w) \leq x_q$, and $\ell_{\min}(\tilde{R}(r_w)) \geq y_q$. Thus, the claim follows if we can show that there are no two slaves $w \neq w'$ with $r_w = r_{w'}$.

So assume for the sake of contradiction that there are two such slaves $w \neq w' \in \mathcal{T}$. Let $w$ be the slave of a node $v$, and let $w'$ be the slave of a node $v'$. Observe that all nodes $z \in \mathcal{T}$ so that $r_w \in T_z$ appear on a path from a leaf of $\mathcal{T}$ to the root. Hence, w.l.o.g., $w$ is an ancestor of $w'$ and $T_{w'} \subseteq T_w$. Let $u'$ be the master of $v'$. Then $p(r_w) \in T_{u'} \subseteq T_{v'} \subseteq T_w$ because $w$ is an ancestor of $v'$. However, since $w$ is itself a slave, $p(r_w) \in T_u$, where $u$ is the master of node $v$. Hence, $p(r_w) \notin T_w$, which leads to the desired contradiction. □

## 12.3 Constructing a Fair Split Tree

Our algorithm for computing a WSPD of a point set $S$ is based on the algorithm of [40]. As this algorithm uses the information provided by a fair split tree of $S$ to derive the desired WSPD of $S$, we first present an I/O-efficient algorithm to compute a fair split tree of a point set. The algorithm follows the framework of the parallel algorithm of [34]; but we do not simulate the PRAM algorithm as this would lead to a higher I/O-complexity. We first outline the algorithm and then show that each of its steps can be carried out I/O-efficiently. As we follow the framework of [34],

**Procedure** FAIRSPLITTREE

**Input:** A point set $S \subset \mathbb{R}^d$ and a box $R_0$ containing all points in $S$.
**Output:** A fair split tree $T$ of $S$.

1: **if** $|S| \leq M$ **then**
2: Load point set $S$ into internal memory and apply the algorithm of [40] to compute $T$.
3: **else**
4: Apply procedure PARTIALFAIRSPLITTREE to compute a partial fair split tree $T'$ of $S$. The leaves of $T'$ have size at most $N^\alpha$.
5: Let $S_1, \ldots, S_k$ be the leaves of $T'$.
6: **for** $i = 1, \ldots, k$ **do**
7: Apply procedure FAIRSPLITTREE recursively to point set $S_i$ and the outer rectangle $\hat{R}(S_i)$ of $S_i$ in $T'$ to compute a fair split tree $T_i$ of $S_i$.
8: **end for**
9: $T = T' \cup T_1 \cup \cdots \cup T_k$.
10: **end if**

**Algorithm 12.4**
Computing a fair split tree of a point set.

the correctness of our algorithm follows from [34], provided that the presented I/O-efficient versions of the different steps of the framework are correct.

To compute a fair split tree $T$ of a point set $S$, we use Algorithm 12.4, providing it with set $S$ and a cube $R_0$ containing $S$ as input. The side length of $R_0$ is $\ell(R_0) = \ell_{\max}(S)$. The centers of $R_0$ and $R(S)$ coincide. The algorithm computes the split tree $T$ recursively. First it computes a partial fair split tree $T'$ of $S$. Then it recursively computes fair split trees for the leaves of $T'$. In the rest of this section, we show that $T'$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. As we show in the next lemma, the leaves of $T'$ are small enough to ensure that the total I/O-complexity of Algorithm 12.4 is $\mathcal{O}(\text{sort}(N))$.

**Lemma 12.10** *Given a set $S$ of $N$ points in $\mathbb{R}^d$, Algorithm 12.4 computes a fair split tree $T$ of $S$ in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space.*

**Procedure** PARTIALFAIRSPLITTREE

**Input:** A point set $S \subset \mathbb{R}^d$ and a box $R_0$ containing all points in $S$.
**Output:** A partial fair split tree $T'$ of $S$ whose leaves have size at most $|S|^\alpha$.

1: Compute a compressed pseudo split tree $T_c$ of $S$ so that none of the leaves of $T_c$ has size more than $|S|^\alpha$.
2: Expand tree $T_c$ to obtain a pseudo split tree $T''$ of $S$ whose leaves have size at most $|S|^\alpha$.
3: Remove all nodes of $T''$ not representing any point in $S$. Compress resulting paths of degree-2 vertices into a single edge. The resulting tree is $T'$.

**Algorithm 12.5**
Computing a partial fair split tree of a point set $S$.

*Proof.* By Lemma 12.11, procedure PARTIALFAIRSPLITTREE correctly computes a partial fair split tree $T'$ of $S$. This implies that Algorithm 12.4 computes a fair split tree of $S$, as it recursively augments $T'$ with fair split trees of its leaves.

By Lemma 12.11, Line 4 of Algorithm 12.4 takes $\mathcal{O}(\text{sort}(N))$ I/Os. Thus, we obtain the following recurrence for the I/O-complexity of Algorithm 12.4:

$$\mathcal{I}(N) \leq c \cdot \text{sort}(N) + \sum_{i=1}^{k} \mathcal{I}(N_i),$$

where $c$ is an appropriate constant so that Algorithm 12.5 takes at most $c \cdot \text{sort}(N)$ I/Os and $N_i$ is the size of set $S_i$. Since $N_i \leq N^\alpha$, this can be expanded to

$$\mathcal{I}(N) \leq c \cdot \text{sort}(N) \sum_{i=0}^{\infty} \alpha^i,$$

whose solution is $\mathcal{I}(N) \leq \frac{c}{1-\alpha}\text{sort}(N) = \mathcal{O}(\text{sort}(N))$. Hence, Algorithm 12.4 takes $\mathcal{O}(\text{sort}(N))$ I/Os. Since Algorithm 12.5 uses linear space, Algorithm 12.4 uses linear space. □

In the rest of this section, we present the details of procedure PARTIALFAIRSPLITTREE (Algorithm 12.5) and prove the following lemma.

**Lemma 12.11** *Given a set $S$ of $N$ points in $\mathbb{R}^d$, Algorithm 12.5 takes $\mathcal{O}(\text{sort}(N))$ I/Os and linear space to compute a partial fair split tree $T'$ of $S$ each of whose leaves represents a subset of at most $N^\alpha$ points of $S$, where $\alpha = 1 - \frac{1}{2d}$.*

Algorithm 12.5 computes the desired partial fair split tree $T'$ in three phases. The first two phases produce a tree $T''$ which is almost a partial fair split tree, except that some of its leaves may represent boxes that are empty. We call $T''$ a *pseudo split tree*. The third phase removes these empty leaves and contracts $T''$ to obtain $T'$.

To construct $T''$, the first phase constructs a tree $T_c$ which is a compressed version of $T''$. In particular, some leaves of $T''$ are missing in $T_c$, and some edges of $T_c$ have to be expanded to a tree that can be obtained by a sequence of fair splits. As we will see, attaching the missing leaves is easy, and the compressed edges represent a particularly nice sequence of fair splits, so that these edges can be expanded I/O-efficiently. Next we provide the details of the three phases of Algorithm 12.5.

### 12.3.1 Constructing $T_c$

To construct tree $T_c$, we partition each dimension of rectangle $R_0$ into slabs so that each slab contains at most $N^\alpha$ points. We ensure that every leaf of $T_c$ is contained in a single slab in at least one dimension. This guarantees that every leaf of $T_c$ contains at most $N^\alpha$ points. The construction of these slabs is the only place where the construction of $T_c$ depends on the point set $S$. Once the slabs are given, we consider the nodes of $T_c$ as rectangles produced from rectangle $R_0$ using fair splits.[1] The slabs are bounded by $\lceil N^{1-\alpha} \rceil + 1$ axes-parallel hyperplanes. We use these slab boundaries to guide the splits in the construction of $T_c$, thus limiting the number of rectangles that can appear as nodes of $T_c$. This is important for the following reason: Even though we introduce the concepts of the construction of $T_c$ as if we constructed $T_c$ by recursively splitting smaller and smaller rectangles, we cannot afford to apply this sequential approach, as it is not I/O-efficient. Instead, we generate all rectangles that might be nodes of $T_c$ and construct a graph which has $T_c$ as a subgraph. Then

[1] The splits are not fair in the exact sense of the definition because it is not guaranteed that there is at least one point on each side of the splitting hyperplane. All other conditions of a fair split are satisfied.

we extract $T_c$ from this graph. The bounded number of nodes guarantees that the constructed graph has linear size, so that we can extract $T_c$ efficiently.

Now consider a node $R \in T_c$ which is to be split. Let $R'$ denote the largest rectangle contained in $R$ whose sides lie on slab boundaries. We maintain the following invariants for all rectangles $R$ generated by the algorithm:

(i) In each dimension, at least one side of $R$ lies on a slab boundary,

(ii) For all $i \in [1, d]$, either $\ell_i(R') = \ell_i(R)$ or $\ell_i(R') \leq \frac{2}{3}\ell_i(R)$, and

(iii) $\ell_{\min}(R) \geq \frac{1}{3}\ell_{\max}(R)$ (i.e., $R$ is a box).

Invariants (i) and (ii) hold for rectangle $R_0$ by definition. Invariant (iii) also holds for $R_0$ because $R_0$ is either a cube or a rectangle computed by a previous invocation of the algorithm. Given that rectangle $R$ satisfies Invariants (i)–(iii), we split it into two rectangles using a hyperplane perpendicular to dimension $i_{\max}(R)$, applying one of the following cases. Each of these cases is said to "produce" one or two rectangles, which are the children of $R$ in $T_c$ and are subject to recursive splitting if necessary. To ensure that these recursive splits can be carried out, we make sure that the rectangles produced by all three cases satisfy Invariants (i)–(iii). A rectangle $R$ is called *constructible* if it can be produced by applying one of these cases to a rectangle which satisfies Invariants (i)–(iii).

**Case 1.** $\ell_{\max}(R) = \ell_{i_{\max}(R)}(R')$. That is both sides of $R$ in dimension $i_{\max}(R)$ lie on slab boundaries (Figure 12.5). We find the slab boundary in this dimension that comes closest to splitting $R$ into equal halves. If this boundary is at distance at least $\frac{1}{3}\ell_{\max}(R)$ from either side of $R$ in dimension $i_{\max}(R)$, Case 1a, we split rectangle $R$ along this slab boundary (Figure 12.5a). Otherwise, Case 1b, we split rectangle $R$ into two equal halves in dimension $i_{\max}(R)$ (Figure 12.5b). This case produces the two rectangles on both sides of the split.

If rectangle $R$ does not satisfy Case 1, $\ell_{i_{\max}(R)}(R') \leq \frac{2}{3}\ell_{\max}(R)$. That is, only one side of $R$ in dimension $i_{\max}(R)$ lies on a slab boundary. We call this slab boundary $H$.

(a) (b)

**Figure 12.5**
The two subcases of Case 1 of the rectangle splitting rule.

**Case 2.** $\ell_{\max}(R') \geq \frac{4}{27}\ell_{\max}(R)$. If $\ell_{i_{\max}(R)}(R') \geq \frac{1}{3}\ell_{\max}(R)$, Case 2a, we split rectangle $R$ along the hyperplane containing the side of $R'$ furthest away from $H$ (Figure 12.6a). Otherwise, Case 2b, we split rectangle $R$ along a hyperplane at distance $y = \frac{2}{3}\left(\frac{4}{3}\right)^j \ell_{\max}(R')$ from $H$, where $j$ is the unique integer so that $\frac{1}{2}\ell_{\max}(R) < \frac{2}{3}\left(\frac{4}{3}\right)^j \ell_{\max}(R') \leq \frac{2}{3}\ell_{\max}(R)$ (Figure 12.6b). Note that $0 \leq j \leq -\lfloor \log \frac{4}{27} / \log \frac{4}{3} \rfloor$. So there are only $\mathcal{O}(1)$ choices for $j$. This case produces only the rectangle containing $R'$. We reattach the other rectangle as a leaf when constructing $T''$ from $T_c$. The reason for not including this second rectangle as a node of $T_c$ is that it violates Invariant (i) in Case 2b. However, the second rectangle contains at most $N^\alpha$ points, as it is contained in a single slab of rectangle $R_0$. Hence, we can attach it as a leaf of $T''$ without violating the constraint that no leaf of $T''$ contains more than $N^\alpha$ points.

**Case 3.** $\ell_{\max}(R') < \frac{4}{27}\ell_{\max}(R)$. Then $R'$ shares a unique corner with $R$. We construct a cube $C$ containing $R'$ that shares the same corner with $R$ and $R'$ and has side length $\ell(C) = \frac{3}{2}\ell_{\max}(R')$ (Figure 12.7). This case produces the rectangle $C$. The edge between $R$ and $C$ is a *compressed edge* which has to be expanded to obtain a sequence of fair splits that produce $C$ from $R$. Again, the reason for introducing this compressed edge is that the rectangles produced by this sequence of fair splits would violate Invariant (i).

It is shown in [34] that each of the rectangles produced by these three cases satisfies Invariants (i)–(iii). It is also shown in [34] that in all three cases, each of

(a) (b)

**Figure 12.6**
The two subcases of Case 2 of the rectangle splitting rule.

the produced rectangles can be uniquely described using two slab boundaries and a constant amount of information in each dimension. As each dimension contains $\mathcal{O}(N^{1-\alpha})$ slab boundaries, the following lemma holds.

**Lemma 12.12 (Callahan [34])** *There are $\mathcal{O}\left(N^{2d(1-\alpha)}\right)$ constructible rectangles.*

We apply the above splitting rules to obtain tree $T_c$ as follows: The root of tree $T_c$ is rectangle $R_0$. This node has two children, as rectangle $R_0$ always satisfies Case 1. For each of the children, we recursively construct a compressed pseudo split tree until all rectangles have been split into rectangles that are not intersected by any slab boundary in at least one dimension. These rectangles are the leaves of $T_c$. Since a rectangle completely contained in a slab contains at most $N^\alpha$ points, it is guaranteed that no leaf in $T_c$ has size more than $N^\alpha$.

As mentioned above, constructing tree $T_c$ using repeated fair splits does not lead to an I/O-efficient algorithm for constructing $T_c$. Hence, we extract tree $T_c$ from a graph $G$ whose nodes are all the constructible rectangles and which contains $T_c$ as a subgraph. Graph $G$ is defined as follows: There is a directed edge $(R_1, R_2)$ in $G$, where $R_1$ and $R_2$ are two constructible rectangles, if rectangle $R_2$ is produced from rectangle $R_1$ by applying the above rectangle splitting rules. This implies that $T_c$ is the subgraph of $G$ containing all nodes $R$ so that there exists a directed path from $R_0$ to $R$ in $G$.

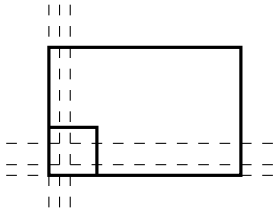


**Figure 12.7**
Case 3 of the rectangle splitting rule.

It is not hard to see that graph $G$ is a DAG, having $R_0$ as one of its sources. Every vertex in $G$ has out-degree at most two, which guarantees that the total size of graph $G$ is linear in the number of constructible rectangles. Choosing $\alpha = 1 - \frac{1}{2d}$, Lemma 12.12 ensures that $G$ has size $\mathcal{O}(N)$. In order to be able to extract $T_c$ from $G$ using the time-forward processing technique, we need an I/O-efficient procedure to topologically sort $G$. This is less trivial than it seems, as there exists no generally I/O-efficient algorithm for topologically sorting sparse graphs. The solution we propose is based on the following observation.

**Observation 12.2** *Let $R_1$ and $R_2$ be the two rectangles produced by splitting a rectangle $R$. Then for $i \in \{1, 2\}$, $\sum_{j=1}^{d} \ell_j(R_i) < \sum_{j=1}^{d} \ell_j(R)$.*

Since graph $G$ contains an edge $(R_1, R_2)$ only if rectangle $R_2$ is one of the rectangles produced by a split of $R_1$ or it is contained in one of the rectangles produced by a split of $R_1$, Observation 12.2 implies that $G$ can be topologically sorted by sorting its vertices by the sums of their side lengths, in decreasing order. We are now ready to present the algorithm for computing $T_c$. The algorithm consists of three steps: Step 1 computes the slabs into which rectangle $R_0$ is partitioned. Step 2 constructs graph $G$ from these slab boundaries. Step 3 extracts tree $T_c$.

**Step 1—Constructing the slabs.** To compute the slabs, we iterate over all $d$ dimensions. For each dimension, we sort the points in $S$ by their coordinates in this dimension. We scan the sorted list of points and place a slab boundary between the

$(jN^\alpha)$-th and $(jN^\alpha + 1)$-st point, for $1 \leq j \leq \lfloor N^{1-\alpha} \rfloor$. In addition, we place slab boundaries which coincide with the sides of $R_0$ perpendicular to the current dimension. As we scan and sort set $S$ once per dimension, the construction of the slabs in all dimensions takes $\mathcal{O}(d \cdot \text{sort}(N)) = \mathcal{O}(\text{sort}(N))$ I/Os.

**Step 2—Constructing graph $G$.** As mentioned before, every constructible rectangle can be encoded using two slab boundaries plus a constant amount of information per dimension. For instance, the $i$-th dimension of a rectangle which is produced by a type-1b split can be fully represented by the two slab boundaries of the original rectangle plus two flags saying that the rectangle is the result of a type-1b split and which of the two produced rectangles this is. Given that rectangles can be represented this way, we construct the vertex set of graph $G$ using $d$ scans, one per list of slab boundaries. The $i$-th scan scans the list of slab boundaries in dimension $i$ and the vertex set of $G$ and fills in the characteristic of each vertex in dimension $i$.

Every node of $G$ now stores a complete representation of the rectangle $R$ it represents. Before constructing the edge set of $G$, we augment every node with a complete description of the largest rectangle $R'$ contained in $R$. To do this, we iterate over all dimensions and compute the boundaries of all rectangles $R'$ in this dimension. To do the latter, we build a buffer tree over the slab boundaries in this dimension. Finding the boundaries of rectangle $R'$ contained in rectangle $R$ now amounts to answering standard search queries on the constructed tree. As there are $\mathcal{O}\left(N^{1-\alpha}\right)$ slab boundaries and $\mathcal{O}(N)$ rectangles $R$, these queries can be answered in $\mathcal{O}(\text{sort}(N))$ I/Os per dimension. As we assume that $d$ is constant, the construction of rectangles $R'$ takes $\mathcal{O}(d \cdot \text{sort}(N)) = \mathcal{O}(\text{sort}(N))$ I/Os in total.

To construct the edge set of $G$, we need to find at most two out-edges, for each rectangle $R$. As the dimensions of rectangles $R$ and $R'$ are stored locally with $R$, we can distinguish between Cases 1, 2, and 3 based only on the information stored with node $R$. The information is also sufficient to distinguish between Cases 2a and 2b, so that in Cases 2 and 3, we can construct the out-edge of rectangle $R$ only from the information stored with $R$. To distinguish between Cases 1a and 1b, we need to find

the slab boundary which comes closest to splitting $R$ in half in dimension $i_{\max}(R)$. To do this, we apply the same approach as for computing rectangles $R'$. In particular, we build a buffer tree over the slab boundaries for each dimension. Then we query this buffer tree with the position of the hyperplane $H'$ splitting rectangle $R$ in half. This query produces the position of the two slab boundaries on each side of the hyperplane, so that it is easy to select the one which is closer to $H'$. Again, this requires $\mathcal{O}(N)$ queries on buffer trees of size $\mathcal{O}\left(N^{1-\alpha}\right)$ to be answered. Hence, the edge set of $G$ can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os.

Instead of adding edges as separate records it is more convenient to represent the edges implicitly by storing all vertices as triples $(R, R_1, R_2)$, where $R_1$ and $R_2$ are the two rectangles produced by the split of $R$. If the split of $R$ produces only one rectangle $R_1$, we represent $R$ by the triple $(R, R_1, \textbf{null})$. If $R$ is a leaf, we represent $R$ by the triple $(R, \textbf{null}, \textbf{null})$.

**Step 3—Extracting $T_c$.** Given graph $G$ as constructed in Step 2 of the algorithm, the extraction of $T_c$ is now rather straightforward. In particular, graph $G$ can be topologically sorted in $\mathcal{O}(\text{sort}(N))$ I/Os, using Observation 12.2. Then we label every node except $R_0$ as inactive. Node $R_0$ is labelled as active. We apply time-forward processing to relabel the nodes of $G$. In particular, every active node remains active and sends "activate" messages to its out-neighbors. An inactive node receiving an "activate" message along one of its in-edges becomes active and forwards the "activate" message along its out-edges. As graph $G$ has size $\mathcal{O}(N)$, this application of time-forward processing takes $\mathcal{O}(\text{sort}(N))$ I/Os. Once the algorithm finishes, every node reachable from $R_0$ is active, while all other nodes are inactive. However, we have observed above that a node is in $T_c$ if and only if it is reachable from $R_0$ in $G$. Hence, a final scan of the vertex set of $G$ suffices to extract the vertices of tree $T_c$. This takes another $\mathcal{O}(\text{scan}(N))$ I/Os.

We have shown that each of the three steps of the construction of $T_c$ can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os. Hence, we obtain the following lemma.

**Lemma 12.13** *Given a point set $S \subset \mathbb{R}^d$ and a rectangle $R_0$ enclosing $S$, a compressed pseudo split tree of $S$ with root $R_0$ can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space.*

### 12.3.2 Constructing $T''$

The goal of the second phase of Algorithm 12.5 is to construct a pseudo split tree $T''$ of $S$ from the compressed pseudo split tree $T_c$ constructed in Phase 1. That is, tree $T''$ may have some leaves that do not correspond to any point in $S$. Apart from that it is a partial fair split tree whose leaves represent point sets of size at most $N^\alpha$. In order to obtain $T''$ from $T_c$, the following things need to be done:

(1) The leaves that are discarded in Case 2 because they violate Invariant (i) need to be reattached. It is easy to see that each such leaf is completely contained in a slab. Hence, it contains at most $N^\alpha$ points, and just attaching it is sufficient. No splits are required.

(2) Every compressed edge $(R, C)$ of $T_c$ produced by an application of Case 3 needs to be replaced by a sequence of fair splits producing the shrunk cube $C$ from rectangle $R$. Note that such a sequence of fair splits exists because $\ell(C) = \frac{2}{3}\ell_{\max}(R') < \frac{2}{9}\ell_{\max}(R) \leq \frac{2}{3}\ell_{\min}(R)$.

(3) The points of $S$ need to be distributed to the leaves of $T''$ containing them. This is required to recognize and discard empty leaves in Phase 3 of Algorithm 12.5. Also, the recursive construction of fair split trees for the leaves of the partial fair split tree obtained at the end of Phase 3 requires the point set contained in each leaf.

Next we show how to carry out these tasks in an I/O-efficient manner.

**Step 1—Reattaching missing leaves.** Recall that every node $R$ in $T_c$ stores a full description of rectangles $R$ and $R'$. As we argued in Section 12.3.1, this is sufficient to distinguish between Cases 1, 2, and 3, and to compute the rectangle produced

by a type-2 split. Given a type-2 rectangle $R$ and the rectangle $R_1$ the type-2 split produces, the discarded rectangle is rectangle $R_2 = R \setminus R_1$.

Given that the information stored with every node $R$ of $T_c$ is sufficient to decide whether it is of type 2 and to compute the rectangle that has been discarded after splitting $R$, a single scan of the vertex set of $T_c$ suffices to produce all discarded nodes. In particular, for every type-2 node $R$ represented by the triple $(R, R_1, \textbf{null})$, we change its triple to $(R, R_1, R_2)$ and add node $R_2$ to the vertex set of $T_c$. This takes $\mathcal{O}(\text{scan}(|T_c|)) = \mathcal{O}(\text{scan}(N))$ I/Os. We call the resulting tree $T_c^+$.

**Step 2—Distributing the points of $S$ and expanding compressed edges.** As a result of the previous step, every point in $S$ is contained either in some box which is a leaf of tree $T_c$, or it is contained in a region $R \setminus C$, where $(R, C)$ is a compressed edge produced by Case 3. Before expanding all compressed edges, we need to determine the region containing each of the points in $S$. This is equivalent to answering deepest containment queries on $T_c^+$, for all points in $S$. By Lemma 12.5, this takes $\mathcal{O}(\text{sort}(N))$ I/Os.

To replace every compressed edge $(R, C)$ produced by Case 3, we simulate one phase of the internal memory algorithm of [40] for constructing a fair split tree. This produces a tree $T(R, C)$ whose leaves form a partition of $R$ into boxes. One of these leaves is $C$. All internal nodes of $T(R, C)$ are ancestors of $C$. That is, intuitively, tree $T(R, C)$ is a path from $R$ to $C$ with an extra leaf attached to every node on this path except $C$. Note that every leaf $R' \neq C$ of $T(R, C)$ contains at most $N^\alpha$ points, as it is completely contained between two slab boundaries in dimension $i_{\max}(p(R'))$. Hence, replacing every edge $(R, C)$ in $T_c^+$ with its corresponding tree $T(R, C)$ produces a pseudo split tree $T''$ of $S$. It remains to show how to construct tree $T(R, C)$ I/O-efficiently.

We apply the following process recursively, starting with $R' = R$, until $R' = C$: If $\ell_{\max}(R') > 3\ell(C)$, we apply a split in dimension $i_{\max}(R')$ to produce two rectangles $R_1$ and $R_2$ of equal size. Otherwise, observe that $R'$ and $C$ share one side perpendicular to dimension $i_{\max}(R')$. Let $H$ be the hyperplane containing the

other side of $C$ perpendicular to dimension $i_{\max}(R')$, and let $R_1$ and $R_2$ be the two rectangles produced by splitting $R'$ along hyperplane $H$. In both cases, let $R_1$ be the rectangle containing $C$. It is not hard to verify that both $R_1$ and $R_2$ are boxes and that either $\ell_i(R_1) = \ell(C)$ or $\ell_i(R_1) \geq \frac{3}{2}\ell(C)$, for every dimension $1 \leq i \leq d$. The latter condition guarantees that the procedure can be applied recursively to rectangle $R_1$. If rectangle $R_2$ contains at least one point, we distribute the points in $R' \setminus C$ to rectangles $R_1$ and $R_2$, make rectangles $R_1$ and $R_2$ children of rectangle $R'$, and repeat the process with $R' = R_1$. Otherwise, we simply replace rectangle $R'$ with $R_1$, essentially shrinking rectangle $R'$, and repeat. This guarantees that only $\mathcal{O}(N)$ splits are performed for all compressed edges $(R, C)$ in $T_c^+$.

To carry out this procedure, we use $d$ sorted lists $L_1, \ldots, L_d$, where list $L_i$ stores the points in $R \setminus C$ sorted by their $i$-th coordinates. When splitting rectangle $R'$ in dimension $i_{\max} = i_{\max}(R')$, we scan list $L_{i_{\max}}$ from the appropriate end until the first point in $R_1$ is found. If this is the first point in $L_{i_{\max}}$, we perform a shrink operation, as rectangle $R_2$ is empty. Otherwise, we remove the scanned points from list $L_{i_{\max}}$ and add them to the point set of leaf $R_2$. Now, in order to invoke the algorithm recursively, the points in $R_2$ have to be removed from all lists $L_i$, $i \neq i_{\max}$. Unfortunately, this may be expensive, as there is no guarantee that these points are stored consecutively in these lists. Hence, we do not modify lists $L_i$, $i \neq i_{\max}$, at this point. Instead, when using such a list $L_i$ to perform a subsequent split, we augment the scan as follows: For every scanned point, we test whether it is contained in $R_2$. If this is the case, we append it to the point set of leaf $R_2$ as above. Otherwise, we remove it from $L_i$ without any further action, as it is contained in $R \setminus R'$ and hence has been written to the point set of some leaf of $T(R, C)$ produced before. In total, every list is scanned at most once, so that $\mathcal{O}(\text{scan}(|S \cap (R \setminus C)|))$ I/Os are sufficient to compute tree $T(R, C)$. Hence, the replacement of all edges $(R, C)$ with trees $T(R, C)$ takes $\mathcal{O}(\text{sort}(N))$ I/Os.

Observe that the resulting tree $T''$ has size $\mathcal{O}(N)$: Tree $T_c$ has size $\mathcal{O}(N)$. In the first step of the construction of $T''$ from $T_c$, every node of $T_c$ gains at most one

child, which is a leaf. The second step adds two nodes per performed split in the construction of a tree $T(R, C)$. As argued above, only $\mathcal{O}(N)$ splits are performed, so that the claim follows. We have shown the following lemma.

**Lemma 12.14** *Given a compressed pseudo split tree $T_c$ of a set $S$ of $N$ points in $\mathbb{R}^d$ as computed in Phase 1 of Algorithm 12.5, Phase 2 takes $\mathcal{O}(\text{sort}(N))$ I/Os and linear space to compute a pseudo split tree $T''$ of $S$ so that every leaf of $T''$ represents at most $N^\alpha$ points in $S$.*

### 12.3.3 Constructing $T'$

Given the pseudo split tree $T''$ computed in Phase 2 of the algorithm, we obtain a partial fair split tree $T'$ from $T''$ by removing all nodes $R$ from $T''$ so that $R \cap S = \emptyset$ and compressing all paths of degree-2 nodes in the resulting tree. Given the list of pairs $(p, R)$, $R \in T''$ and $p \in S \cap R$, produced by the previous two phases, we sort the pairs in this list by their second components. We sort the vertex set of $T''$. Now a single scan of these two sorted lists suffices to mark every leaf $R$ of $T''$ such that $R \cap S = \emptyset$ as "remove" and all remaining leaves as "keep". We apply time-forward processing to process $T''$ from the leaves toward the root to mark every internal node as "keep", "contract", or "remove", depending on whether none, one, or both of its children are marked as "remove". In addition to these marks, we label every node $R$ with its closest descendant $K(R)$ which is labelled "keep". In particular, $K(R) = R$ if $R$ is itself labelled "keep", and $K(R) = K(R')$ if $R$ is labelled "contract", where $R'$ is the child of $R$ not labelled "remove". Finally, for every node labelled "keep", we replace both of its children $R_1$ and $R_2$ with their corresponding descendants $K(R_1)$ and $K(R_2)$. The tree induced by all nodes marked as "keep" is tree $T'$. We scan the vertex set of $T''$ one more time to remove all nodes marked as either "remove" or "contract". All the techniques used in this procedure take $\mathcal{O}(\text{sort}(N))$ I/Os and linear space. Hence, this third phase of Algorithm 12.5 takes $\mathcal{O}(\text{sort}(N))$ I/Os and linear space, so that we obtain the following lemma.

**Lemma 12.15** *Given a pseudo split tree $T''$ of a set $S \subset \mathbb{R}^d$ of $N$ points, as computed in Phase 2 of Algorithm 12.5, Phase 3 takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os and linear space to compute a partial fair split tree $T'$ of $S$ so that every leaf in $T'$ represents at most $N^\alpha$ points in $S$.*

## 12.4 Constructing a WSPD

In this section, we describe an I/O-efficient algorithm to extract a WSPD of a point set $S$ from a fair split tree $T$ of $S$ as constructed by Algorithm 12.4. We assume that every leaf $p$ of $T$ is labelled with the coordinates of point $p$. Every internal node $A$ is labelled with its bounding rectangle $R(A)$. Every pair $\{A_i, B_i\}$ in the computed WSPD is represented as the pair of nodes in $T$ rather than using a full representation of both sets because otherwise the output may have size $\Omega(N^2)$.

Our algorithm simulates the internal memory algorithm of [40], using the fair split tree to drive the generation of pairs. We show that this computation can be translated into a traversal of a DAG $G$ of size $\mathcal{O}(N)$. This traversal can be realized using the time-forward processing technique. The difficulty is that we are not able to generate $G$ beforehand because the presence of edges in $G$ is decided only while the algorithm runs. We could generate a supergraph of $G$ containing all edges that could possibly be in $G$; but there are $\Omega(N^2)$ such edges in the worst case, so that this does not lead to an efficient algorithm. Next we define DAG $G$ and show that it can be generated efficiently while it is being traversed.

In order to define DAG $G$, we need to consider the internal memory algorithm of [40] for computing a well-separated realization of a point set $S$ from its fair split tree. We call this procedure ComputeWSR. Its pseudo-code is shown in Algorithm 12.6. In this algorithm we define $A \prec B$ if either $\ell_{\max}(A) < \ell_{\max}(B)$ or $\ell_{\max}(A) = \ell_{\max}(B)$ and $\nu(A) < \nu(B)$, for some postorder numbering $\nu$ of $T$.

It is not hard to see that the set $R$ constructed in Line 1 of procedure ComputeWSR is a realization of $S \otimes S$. However, this realization may not be well-separated. Now the algorithm iterates over all pairs in $R$ and tests whether they

**Procedure** ComputeWSR

**Input:** A point set $S \subset \mathbb{R}^d$ and a fair split tree $T$ of $S$.
**Output:** A well-separated realization $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ of $S \otimes S$ consisting of $k = \mathcal{O}(|S|)$ pairs.

For every internal node of $T$ with children $A$ and $B$, add a pair $\{A, B\}$ to a set $R$.
$\mathcal{R} \leftarrow \emptyset$
**for** every pair $\{A, B\} \in R$ **do**
  Remove pair $\{A, B\}$ from $R$.
  $\mathcal{R} \leftarrow \mathcal{R} \cup$ FindPairs$(\{A, B\}, T)$
**end for**

**Procedure** FindPairs

**Input:** A fair-split tree $T$ and a pair $\{A, B\}$ of siblings in $T$.
**Output:** A well-separated realization $\mathcal{R}'$ of $A \otimes B$.

$R' \leftarrow \{\{A, B\}\}$
$\mathcal{R}' \leftarrow \emptyset$
**while** $R'$ is not empty **do**
  Remove a pair $\{A, B\}$ from $R'$.
  {Assume w.l.o.g. that $B \prec A$.}
  **if** $A$ and $B$ are well-separated **then**
    Add pair $\{A, B\}$ to $\mathcal{R}'$.
  **else**
    Let $A_1$ and $A_2$ be the two children of $A$ in $T$.
    Add pairs $\{A_1, B\}$ and $\{A_2, B\}$ to $R'$.
  **end if**
**end while**

**Algorithm 12.6**
Computing a well-separated realization from a fair split tree.

are well-separated. If a pair $\{A, B\}$ is well-separated, it is added to realization $\mathcal{R}$. Otherwise, it is replaced by a well-separated realization of $A \otimes B$. This way, the algorithm maintains the invariant that $R \cup \mathcal{R}$ is a realization of $S \otimes S$ and all pairs in $\mathcal{R}$ are well-separated. As set $R$ is empty when the algorithm terminates, the final set $\mathcal{R}$ is a well-separated realization of $S \otimes S$. Using packing arguments, Callahan and Kosaraju [40] show that the computed realization has size $\mathcal{O}(|S|)$. To show that the algorithm takes $\mathcal{O}(|S|)$ time to produce realization $\mathcal{R}$, they introduce the concept of a *computation tree*. Such a tree represents the recursive invocations of procedure FindPairs triggered by an invocation of procedure FindPairs in Line 5 of procedure ComputeWSR.

Formally, a computation tree is defined as follows: Let $\{A, B\}$, $B \prec A$, be a pair of nodes in $T$. If $A$ and $B$ are well-separated, the computation tree $T(\{A, B\})$ has a single node $(A, B)$. Otherwise, tree $T(\{A, B\})$ consists of two computation trees $T(\{A_1, B\})$ and $T(\{A_2, B\})$ whose roots are the children of the root $(A, B)$ of $T(\{A, B\})$, where $A_1$ and $A_2$ are the children of $A$ in $T$. The size of a computation tree is proportional to the number of its leaves. Each of these leaves corresponds to a pair in the computed well-separated realization, so that the total size of all computation trees $T(\{A, B\})$, $\{A, B\} \in R$, is $\mathcal{O}(|S|)$.

We are now ready to describe the DAG $G$ so that the computation of procedure ComputeWSR can be simulated using a traversal of $G$. The vertex set of $G$ is the same as that of the given fair split tree $T$. There is an edge from a node $A_1$ to a node $A_2$ in $G$ if there are two nodes $(A_1, B_1)$ and $(A_2, B_2)$ in a computation tree $T(\{A, B\})$, $\{A, B\} \in R$, so that $(A_1, B_1)$ is the parent of $(A_2, B_2)$. Next we show that $G$ is indeed a DAG and that the computation of procedure ComputeWSR corresponds to some traversal of graph $G$.

**Lemma 12.16** *Graph $G$ is a DAG.*

*Proof.* We show that for every edge $(A_1, A_2)$ in $G$, $A_2 \prec A_1$. Moreover, it is easy to verify that "$\prec$" is a total order. This implies that $G$ is acyclic. To see that $A_2 \prec A_1$, observe that edge $(A_1, A_2)$ is in $G$ only because there is an edge $((A_1, B_1), (A_2, B_2))$

in a computation tree. This, however, implies that either $A_2 = B_1$ or $A_2$ is a child of $A_1$ in $T$. In the former case, $A_2 \prec A_1$, by definition. In the latter case, $\ell_{\max}(A_2) \leq \ell_{\max}(A_1)$ and $\nu(A_2) < \nu(A_1)$, so that again $A_2 \prec A_1$. □

The computation of procedure ComputeWSR can be simulated by a traversal of $G$ as follows: Initially, we store every pair $\{A, B\} \in R$, $B \prec A$, with node $A \in G$. Then we process the nodes of $G$ in topologically sorted order. For every node $A \in G$, we process the pairs stored with node $A$ one by one. For every pair $\{A, B\}$, $B \prec A$, we check whether $A$ and $B$ are well-separated. If so, we add the pair to the well-separated realization. Otherwise, let $(A', B')$ and $(A'', B'')$ be the children of node $(A, B)$ in the computation tree containing node $(A, B)$. We "send" pair $\{A', B'\}$ along edge $(A, A')$ and pair $\{A'', B''\}$ along edge $(A, A'')$ to add these pairs to the sets of pairs stored with these two nodes. By the definition of $G$, edges $(A, A')$ and $(A, A'')$ exist because edges $((A, B), (A', B'))$ and $((A, B), (A'', B''))$ exist in the computation tree.

Next we argue that the edge set of graph $G$ does not have to be known in advance in order to perform the above traversal of $G$ using time-forward processing. To prepare graph $G$, we compute a postorder numbering of the nodes of $T$. We sort the nodes of $T$ by the relation "$\prec$" defined by this postorder numbering and assign a number $\eta(A)$ to every node $A \in T$, which represents the position of node $A$ in the sorted sequence of nodes. We store with every node $A$ of $T$ the labels $\eta(A_1)$ and $\eta(A_2)$ of its two children $A_1$ and $A_2$ in $T$. Finally, we sort the nodes of $T$ by decreasing numbers $\eta(A)$. This preprocessing can be carried out in $\mathcal{O}(\mathrm{sort}(|S|))$ I/Os, as computing a postorder numbering and copying the label $\eta(A)$ of each node to its parent can be realized using time-forward processing. Besides that, we sort the vertex set of $T$ twice.

To initiate the processing of $G$, we scan the list of nodes. For every internal node of $T$ with children $B \prec A$, we insert the pair $(A, B)$ into a priority queue $Q$ and give it priority $\eta(A)$. Now we process the nodes of $G$ in their order of appearance. For each node $A$, we retrieve all pairs $(A, B)$ from $Q$, one by one. For every pair $(A, B)$, we test whether $A$ and $B$ are well-separated. If so, we add pair $\{A, B\}$ to the realization. Otherwise, let $A_1$ and $A_2$ be the two children of $A$. (Recall that numbers

$\eta(A_1)$ and $\eta(A_2)$ are stored with node $A$.) If $\eta(A_1) > \eta(B)$, we insert pair $(A_1, B)$ into priority queue $Q$ and give it priority $\eta(A_1)$. Otherwise, we insert pair $(B, A_1)$ into priority queue $Q$ and give it priority $\eta(B)$. The same is done for child $A_2$. This is equivalent to sending pairs $\{A_1, B\}$ and $\{A_2, B\}$ along the corresponding edges of DAG $G$. Then we proceed to the next pair.

The procedure just described is the same as standard time-forward processing with the exception that every node constructs its out-neighborhood depending on the data it receives from its in-neighbors. The crucial fact is that the constructed out-neighborhood of a node $A$ contains only nodes $B$, $\eta(B) < \eta(A)$, so that every node sending a pair to node $B$ does so before node $B$ is being processed by the above procedure.

All that remains to be done is bound the number of priority queue operations performed. Note that every pair $(A, B)$ corresponding to a node in a computation tree is queued and dequeued exactly once. Hence, the total number of priority queue operations is at most twice the total size of all computation trees, which is $\mathcal{O}(|S|)$. Hence, simulating procedure ComputeWSR using a traversal of graph $G$ takes $\mathcal{O}(\text{sort}(|S|))$ I/Os. By Lemma 12.10, a fair split tree of a point set $S$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, so that we obtain the following result.

**Theorem 12.2** *Given a set $S$ of $N$ points in $\mathbb{R}^d$, a well-separated pair decomposition of $S$ consisting of $\mathcal{O}(N)$ pairs can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os using linear space.*

## 12.5 Applications of the WSPD

Using Theorem 12.2, we obtain I/O-efficient solutions to a number of proximity problems in $d$-dimensions. In particular, we show that a $t$-spanner of a point set $S$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os and that the $K$-nearest neighbor and $K$-closest pair problems for $S$ can be solved in $\mathcal{O}(\text{sort}(KN))$ and $\mathcal{O}(\text{sort}(N+K))$ I/Os, respectively. The solutions are relatively simple adaptations of the results presented in [35].

### 12.5.1 Computing a $t$-Spanner

In [40], it is shown that the following graph $G = (S, E)$ is a $t$-spanner of linear size for a point set $S \subset \mathbb{R}^d$: Given a WSPD $\mathcal{D} = (T, \mathcal{R})$ of $S$ with separation $s = 4\frac{t+1}{t-1}$ and consisting of $\mathcal{O}(|S|)$ pairs, choose a representative $r(A) \in A$, for every point $A \in T$. For every pair $\{A_i, B_i\}$ in the WSPD, add an edge $\{r(A_i), r(B_i)\}$ to $E$.

To choose representatives for all nodes $A \in T$, we apply time-forward processing in $T$ from the leaves toward the root. Every leaf has itself as a representative. Every internal node chooses one of the representatives of its two children as a representative. Hence, the representatives for all nodes $A \in T$ can be computed in $\mathcal{O}(\text{sort}(|S|))$ I/Os. In order to create the edge set of spanner $G$, we consider every pair $\{A, B\}$ in the WSPD as an edge between nodes $A$ and $B$. Now we apply operation CopyVertexLabels to replace every pair $\{A, B\}$ by the edge $\{r(A), r(B)\}$. This takes $\mathcal{O}(\text{sort}(|S|))$ I/Os.

**Theorem 12.3** *Given a set $S$ of $N$ points in $\mathbb{R}^d$, a $t$-spanner of linear size for $S$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space.*

In [19], it is shown that one can construct a spanner of diameter at most $2 \log N$ by choosing representatives $r(A)$ above more carefully. In particular, given the representatives $r(A_1)$ and $r(A_2)$ for the children $A_1$ and $A_2$ of $A$, we choose $r(A)$ to be the representative $r(A_i)$ of the heavier subtree $T(A_i)$, where the weight of a tree is the number of leaves in the tree. This modified rule for choosing representatives can easily be incorporated in the time-forward processing step used to compute representatives. Hence, we obtain the following result.

**Theorem 12.4** *Given a set $S$ of $N$ points in $\mathbb{R}^d$, a $t$-spanner of linear size and spanner diameter at most $2 \log N$ for $S$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space.*

Finally, the following result shows that Theorems 12.3 and 12.4 are optimal.

**Theorem 12.5** *It takes $\Omega(\text{sort}(N))$ I/Os to compute a $t$-spanner of linear size for a set $S$ of $N$ points in $\mathbb{R}^d$.*

*Proof.* Given a set $S = \{p_1, \ldots, p_N\}$ of points in $\mathbb{R}^d$, a $t$-spanner $G$ of $S$ has to contain an edge $\{p_i, p_j\}$, for every pair of points $p_i = p_j$, $i \neq j$. Hence, a single scan of the edge set of $G$ is sufficient to decide whether all points in $S$ are distinct. If $G$ has linear size, this takes $\mathcal{O}(\text{scan}(N))$ I/Os. Thus, if $G$ can be constructed in $o(\text{sort}(N))$ I/Os, the element uniqueness problem can be solved in $o(\text{sort}(N))$ I/Os, which contradicts the $\Omega(\text{sort}(N))$ I/O lower bound shown for this problem in [14]. □

### 12.5.2 $K$-Nearest Neighbors

In this section, we show how to compute the $K$ nearest neighbors for every point $p$ of a point set $S \subset \mathbb{R}^d$, i.e., the $K$ points in $S \setminus \{p\}$ which are closest to $p$. The construction follows the internal memory algorithm of [40] for this problem.

**Lemma 12.17 (Callahan/Kosaraju [40])** *Let $\{A, B\}$ be a pair in a well-separated realization of $S \otimes S$ with separation $s > 2$. If there is a point $b \in B$ that is among the $K$ nearest neighbors of a point $a \in A$, then $|A| \leq K$.*

Given a set $B$, let $O_B$ be the center of its bounding rectangle $R(B)$. Then divide the space around $B$ into a constant number of cones with apex $O_B$ such that for any two points $a$ and $a'$ in the same cone, the angle $\angle aO_Ba'$ between segments $\overline{O_Ba}$ and $\overline{O_Ba'}$ is less than $\alpha = \frac{s}{s+1}$. The existence of such a set of cones has been shown by Yao [176], who calls such a set of cones an *$\alpha$-frame*.

**Lemma 12.18 (Callahan/Kosaraju [40])** *Let $X$ and $B$ be point sets such that for every point $x \in X$, the pair $\{\{x\}, B\}$ is well-separated. Let $C$ be a cone with apex $O_B$ such that for any two points $a$ and $a'$ in $C$, $\angle aO_Ba' < \frac{s}{s+1}$. Let $X \cap C = (x_1, \ldots, x_l)$ be the set of points in $X$ that lie in $C$, sorted by their distances from $O_B$. For $i > K$, no point in $B$ can be among the $K$ nearest neighbors of $x_i$.*

Based on Lemma 12.17, the algorithm of [40] first extracts all pairs $\{A_i, B_i\}$ with $|A_i| \leq K$. For a node $B$ in $T$, let $\{A'_1, B\}, \ldots, \{A'_q, B\}$ be the set of pairs in the WSPD that contain $B$ and such that $|A'_i| \leq K$. Note that sets $A'_1, \ldots, A'_q$

are pairwise disjoint. The algorithm constructs a candidate set $X(B) = \bigcup_{i=1}^{q} A'_i$, for every node $B \in T$. Now the nodes of $T$ are processed from the root toward the leaves. For every node $B \in T$, the algorithm creates an $\alpha$-frame $\mathcal{C}$ around $O_B$, partitions the points in $X(B)$ into sets $X_C = X(B) \cap C$, $C \in \mathcal{C}$, and extracts the set $X'_C$ of the $K$ points in $X_C$ closest to $O_B$. Let $N(B) = \bigcup_{C \in \mathcal{C}} X'_C$. From the two lemmas above it follows that $N(B)$ contains all points $p \in S$ which have one of their $K$ nearest neighbors in $B$. Now the candidate sets $X(B_1)$ and $X(B_2)$ of the two children $B_1$ and $B_2$ of $B$ in $T$ are augmented with the points in $N(B)$. Then the algorithm proceeds to the next node. Eventually, the procedure produces sets $N(b)$, $b \in S$, so that the points having $b$ as one of their $K$ nearest neighbors are contained in $N(b)$. The algorithm produces sets $N'(a)$, $a \in S$, so that $b \in N'(a)$ if and only if $a \in N(b)$. Finally, the $K$ nearest neighbors of every point $a$ are extracted from $N'(a)$ using $K$-selection.

The crucial observation used in [40] to show that this algorithm takes $\mathcal{O}(KN)$ time is the following: Initially, the candidate sets $X(B)$, $B \in T$, have total size $\mathcal{O}(KN)$ because there are only $\mathcal{O}(N)$ pairs in the given WSPD, and every pair contributes at most $K$ points to some set $X(B)$. When processing $T$ top-down, each of the $\mathcal{O}(N)$ sets $X(B)$ is augmented with the $\mathcal{O}(K)$ points in $N(p(B))$, which adds another $\mathcal{O}(KN)$ points to the total size of all candidate sets $X(B)$. Hence, during the construction of sets $N(b)$, $K$-selection is applied to candidate sets of total size $\mathcal{O}(KN)$, which takes $\mathcal{O}(KN)$ time. In internal memory, sets $N'(a)$ are readily constructed in time $\mathcal{O}\left(\sum_{b \in S} |N(b)|\right) = \mathcal{O}(KN)$ time, and the final application of $K$-selection to these sets takes $\mathcal{O}(KN)$ time again.

Once the initial candidate sets $X(B)$, $B \in T$, have been computed, the rest of the algorithm can easily be carried out in $\mathcal{O}(\text{sort}(KN))$ I/Os. In particular, we realize the augmentation of every set $X(B)$ with the points in $N(p(B))$ using time-forward processing, sending the content of set $N(p(B))$ from $p(B)$ to $B$. When processing node $B$, we append the received points to $X(B)$ and apply an I/O-efficient

$K$-selection algorithm to $X(B)$, which takes $\mathcal{O}(\text{scan}(|X(B)|))$ I/Os[2]. Hence, the total number of I/Os spent on constructing sets $N(b)$, for all leaves $b$ of $T$, is $\mathcal{O}(\text{sort}(KN))$: $\mathcal{O}(\text{sort}(KN))$ I/Os to send sets $N(p(B))$ along the edges of $T$ using time-forward processing, and $\mathcal{O}(\text{scan}(KN))$ I/Os for all applications of $K$-selection.

Given that every set $N(b)$ is represented as a collection of pairs $\{(a, b) : a \in N(b)\}$, where $b$ is a leaf of $T$, sets $N'(a)$, $a \in S$, can be constructed by sorting the union of these sets lexicographically. Then we apply $K$-selection to each of these sets $N'(a)$ to extract the $K$ nearest neighbors of point $a$. This takes another $\mathcal{O}(\text{sort}(KN))$ I/Os. It remains to show how to construct the initial candidate sets $X(B)$, $B \in T$, I/O-efficiently.

First we compute a postorder numbering $\nu$ of the nodes of $T$ and a labelling of every node $A \in T$ with the number $\lambda(A)$ of leaves that are descendants of $A$ in $T$. We apply procedure CopyVertexLabels to inform every pair $\{A, B\}$ in the WSPD about the postorder numbers $\nu(A)$ and $\nu(B)$ of its endpoints. Then we create two copies of every pair $\{A, B\}$ in the given WSPD, representing one as the ordered pair $(A, B)$ and the other as the ordered pair $(B, A)$. We sort the nodes of $T$ according to the above postorder numbering and the ordered pairs $(A, B)$ and $(B, A)$ by the postorder numbers of their first components. We scan the sorted list of nodes and create a list $L$ containing all leaves of $T$, sorted by increasing postorder numbers. Now we scan the list of nodes of $T$, the list of pairs in the WSPD, and the list of leaves of $T$. The scans of the latter two lists are driven by the information found in the node list as follows: For every node $A \in T$, we continue the scan of list $L$ until a leaf $a$ with $\nu(a) > \nu(A)$ is found. If $\lambda(A) > K$, we skip over all pairs $(A, B)$ in the list of pairs. If $\lambda(A) \leq K$, we repeat the following for every pair $(A, B)$ in the list of pairs: We scan backward from the current position in $L$ to read the last $\lambda = \lambda(A)$ leaves $v_1, \ldots, v_\lambda$ in $L$ and append pairs $(\nu(B), v_1), \ldots, (\nu(B), v_\lambda)$ to a list $\mathcal{X}$. Once all

[2]It is easy to verify that the standard $K$-selection algorithm of [50] takes $\mathcal{O}(\text{scan}(N))$ I/Os when applied to a set of size $N$.

nodes of $T$ have been processed, we sort the pairs in list $\mathcal{X}$ by their first components, thereby transforming list $\mathcal{X}$ into a concatenation of lists $X(B)$, $B \in T$.

The computation of labels $\nu(A)$ and $\lambda(A)$, for all nodes $A \in T$, can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os, using time-forward processing. The application of procedure CopyVertexLabels takes $\mathcal{O}(\text{sort}(N))$ I/Os. Next the algorithm sorts three lists of size $\mathcal{O}(N)$, which takes $\mathcal{O}(\text{sort}(N))$ I/Os. The scans used to produce list $\mathcal{X}$ scan a total of $\mathcal{O}(KN)$ data items, so that this takes $\mathcal{O}(\text{scan}(KN))$ I/Os. The final sort of list $\mathcal{X}$ takes $\mathcal{O}(\text{sort}(KN))$ I/Os because list $\mathcal{X}$ has size $\mathcal{O}(KN)$. Hence, we obtain the following result.

**Theorem 12.6** *Given a set $S$ of $N$ points in $\mathbb{R}^d$, the $K$ nearest neighbors of every point in $S$ can be computed in $\mathcal{O}(\text{sort}(KN))$ I/Os using $\mathcal{O}(KN/B)$ blocks of external memory.*

### 12.5.3 $K$-Closest Pairs

In this section, we show how to enumerate the $K$ smallest interpoint distances in a point set $S \subset \mathbb{R}^d$. In [38], the following approach has been proposed to solve this problem: Given a WSPD $\mathcal{D}$ of $S$, let $\langle\{A_1, B_1\}, \ldots, \{A_q, B_q\}\rangle$ be the list of pairs in $\mathcal{D}$, sorted by increasing distances $\text{dist}_2(R(A_i), R(B_i))$. Find the smallest index $i$ so that $\sum_{j=1}^{i} |A_j||B_j| \geq K$ and retrieve all pairs $\{A, B\}$ such that $\text{dist}_2(R(A), R(B)) \leq (1+4/s)r$, where $r = \text{dist}_2(R(A_i), R(B_i))$. Then generate the set $C$ of all pairs $\{a, b\}$ such that $a \in A$ and $b \in B$, for some pair $\{A, B\}$ with $\text{dist}_2(R(A), R(B)) \leq (1+4/s)r$. Now apply $K$-selection to find the set $X$ of $K$ pairs so that $\text{dist}_2(a, b) \leq \text{dist}_2(a', b')$, for any $\{a, b\} \in X$ and $\{a', b'\} \in C \setminus X$.

The correctness of this solution has been shown in [38]. It is also shown in [38] that set $C$ has size $\mathcal{O}(N + K)$. Thus, once set $C$ has been produced, the application of $K$-selection to $C$ takes $\mathcal{O}(\text{scan}(N+K))$ I/Os. We show that set $C$ can be produced in $\mathcal{O}(\text{sort}(N + K))$ I/Os.

First we use time-forward processing to process $T$ from the leaves toward the root, computing for every node $A \in T$, the size $|A|$ of point set $A$. We use procedure CopyVertexLabels to inform pairs $\{A, B\}$ about the cardinalities and bounding rectangles of sets $A$ and $B$. Now we sort these pairs by their distances $\text{dist}_2(R(A), R(B))$. We scan the list of pairs by increasing distances and sum the cardinalities $|A||B|$ of the scanned pairs $\{A, B\}$. We stop as soon as this sum is at least $K$. Let $\{A_i, B_i\}$ be the pair where the scan stopped. Now we continue the scan until we find the first pair $\{A, B\}$ with $\text{dist}_2(R(A), R(B)) > (1 + 4/s)\text{dist}_2(R(A_i), R(B_i))$. We discard this pair as well as all remaining pairs in the list.

Now we make two copies of each remaining pair $\{A, B\}$, representing one as the ordered pair $(A, B)$ and the other as the ordered pair $(B, A)$. We sort these ordered pairs by their first components, thereby producing for every node $A \in T$, the list of pairs $(A, B_1), \ldots, (A, B_{\rho(A)})$ among the remaining pairs. We use a similar procedure as the one described in Section 12.5.2 to extract for every node $A$ with $\rho(A) > 0$, the list of points in set $A$. For every pair $(A, B_i)$, we make a copy of point set $A$, representing every point $a \in A$ as the triple $(\{A, B_i\}, A, a)$. We sort the resulting list of triples lexicographically. As a result, for every pair $\{A, B\}$, point sets $A$ and $B$ are stored consecutively. A nested scan of these two sets is sufficient to create set $\{\{a, b\} : a \in A \text{ and } b \in B\}$.

The extraction of all relevant pairs takes $\mathcal{O}(\text{sort}(N))$ I/Os, as it involves the application of time-forward processing and procedure CopyVertexLabels to graphs of size $\mathcal{O}(N)$, and the sorting and scanning of a list of $\mathcal{O}(N)$ pairs. Given the extracted pairs, producing lists $(A, B_1), \ldots, (A, B_{\rho(A)})$, for all nodes $A \in T$, takes $\mathcal{O}(\text{sort}(N))$ I/Os, as there are only $\mathcal{O}(N)$ pairs to be sorted. Using the same arguments as in Section 12.5.2, the I/O-complexity of the extraction of pairs of sets $\{A, B\}$ can be bounded by $\mathcal{O}(\text{sort}(N + K))$ because the cardinality of all extracted sets is $\mathcal{O}(N + K)$. Finally, the scan to produce the candidate set of point pairs takes $\mathcal{O}(\text{scan}(N + K))$ I/Os, as $\mathcal{O}(N + K)$ is a bound on the cardinality of the produced set. Hence, we obtain the following result.

**Theorem 12.7** *Given a set $S$ of $N$ points in $\mathbb{R}^d$, the $K$ closest pairs in $S$ can be found in $\mathcal{O}(\text{sort}(N+K))$ I/Os using $\mathcal{O}((N+K)/B)$ blocks of external memory.*

# Chapter 13

# The Dumbbell Spanner

In the previous chapter, we have shown that a well-separated pair decomposition of a point set $S$ can be computed I/O-efficiently. We have presented I/O-efficient algorithms which use the computed WSPD to solve classical proximity problems and construct a $t$-spanner with spanner diameter $\mathcal{O}(\log N)$ for point set $S$. However, in general, spanners are useful only if spanner paths between two query points can be reported efficiently. For the WSPD-spanner, no I/O-efficient query procedure is known.

In this chapter, we show that the dumbbell spanner of [18] can be constructed I/O-efficiently. This spanner is a supergraph of the WSPD-spanner with the desirable property that it can be decomposed into a constant number of trees so that for any two points $p, q \in S$, there is a spanner path between $p$ and $q$ which is contained in one of these trees. Thus, existing solutions for reporting paths in trees [100, 177] can be used to report spanner paths I/O-efficiently.

In Section 13.1, we discuss the concept of dumbbell trees as introduced in [18]. To motivate this construction, we show in Section 13.2 that a spanner for a point set $S \subseteq \mathbb{R}^d$ can be computed I/O-efficiently from the set of dumbbell trees defined by a WSPD of $S$. We show that the dumbbell trees can be used to report spanner paths in the constructed spanner in an I/O-efficient manner. In Section 13.3, we show how to compute dumbbell trees for a given WSPD I/O-efficiently.

## 13.1 Dumbbell Trees

Let $S$ be a set of points in $\mathbb{R}^d$ and let $\mathcal{D} = (T, \mathcal{R})$ be a WSPD of $S$. Let $C$ be the outer rectangle of the root of $T$, which is a cube. For a well-separated pair $\{A, B\} \in \mathcal{R}$, we refer to the set $R(A) \cup R(B)$ as the *dumbbell* $D(A, B)$. Rectangles $R(A)$ and $R(B)$ are the *heads* of dumbbell $D(A, B)$. The *length* $\ell(D(A, B))$ of $D(A, B)$ is defined as the length of the line segment connecting the two centers of $R(A)$ and $R(B)$. Also, we refer to $C$ as a head (which does not belong to any dumbbell). The set of dumbbell trees is computed from a partition of the set of dumbbells into a constant number of sets $\mathcal{G}_1, \ldots, \mathcal{G}_q$ such that the dumbbells in each set have the following three properties, where $0 < \delta < 1/2$ and $c > 0$ are constants:

**Length grouping property:** The lengths of two dumbbells in the same group are either within a factor of two from each other or differ by a factor of at least $1/\delta$.

**Empty region property:** Let $D_1$ and $D_2$ be two dumbbells in the same group so that $\ell(D_1) \leq \ell(D_2) \leq 2\ell(D_1)$. Then the heads of $D_1$ and $D_2$ have distance at least $c\ell(D_1)$ from each other.

**Nesting property:** Two dumbbells in the same group are either completely disjoint, or at least one head of the smaller dumbbell is completely contained in one head of the larger dumbbell.[1]

A dumbbell tree $T_{\mathcal{G}_h}$ for each such group $\mathcal{G}_h$, $1 \leq h \leq q$, of dumbbells is defined as follows. The nodes of $T_{\mathcal{G}_h}$ are of three different types:

**Leaves:** There is one leaf in $T_{\mathcal{G}_h}$ for every point $p \in S$.

**Dumbbell nodes:** There is one dumbbell node in $T_{\mathcal{G}_h}$ for every dumbbell in $\mathcal{G}_h$.

[1] Although the definition of the nesting property in [18] is slightly different, it is equivalent to our definition, given that all dumbbells in a group have the length grouping and empty region properties.

**Head nodes:** For every dumbbell $D(A, B) \in \mathcal{G}_h$, there are two head nodes $R_B(A)$ and $R_A(B)$ in $T_{\mathcal{G}_h}$ which represent the two heads $R(A)$ and $R(B)$ of $D(A, B)$. There is a head node representing the special head $C$.

Similar to the description of the fair split tree in Section 12.3, we do not distinguish between the nodes of $T_{\mathcal{G}_h}$ and the geometric objects they represent. Hence, it makes sense to talk for instance about containment of a leaf of $T_{\mathcal{G}_h}$, which is a point, in a head node, which is a rectangle. Using this convention, the edge set of $T_{\mathcal{G}_h}$ is defined as follows: Partition group $\mathcal{G}_h$ into its subgroups $\mathcal{G}_{h,1}, \ldots, \mathcal{G}_{h,t}$ so that the dumbbells in each group $\mathcal{G}_{h,i}$ differ by a factor of at most two in length. Sort groups $\mathcal{G}_{h,1}, \ldots, \mathcal{G}_{h,t}$ by the lengths of their dumbbells. Let $\mathcal{G}_{h,t+1}$ be another group which contains a dummy dumbbell one of whose heads is $C$.

For every leaf $p$ of $T$, let $i$ be the smallest index so that there is a dumbbell $D(A, B) \in \mathcal{G}_{h,i}$ containing point $p$. Then node $p$ is the child of head node $R_B(A)$ or $R_A(B)$, whichever contains point $p$. Similarly, for every dumbbell $D(A, B) \in \mathcal{G}_{h,i}$, $1 \leq i \leq t$, let $i' > i$ be the smallest index so that there is a dumbbell $D(A', B') \in \mathcal{G}_{h,i'}$ containing one of the heads $R_B(A)$ or $R_A(B)$ of $D(A, B)$. Assume w.l.o.g. that head $R_{B'}(A')$ contains head $R_B(A)$. Then dumbbell node $D(A, B)$ is the child of head node $R_{B'}(A')$. Finally, for every dumbbell $D(A, B) \in \mathcal{G}_{h,i}$, $1 \leq i \leq t$, head nodes $R_B(A)$ and $R_A(B)$ are the children of dumbbell node $D(A, B)$. Following this construction, every node in $T_{\mathcal{G}_h}$ except $C$ has a well-defined parent, so that $C$ is the root of $T_{\mathcal{G}_h}$.

In order to motivate this construction, we show in the next section that a $t$-spanner $G$ with spanner diameter $\mathcal{O}(\log N)$ can be constructed in an I/O-efficient manner from the set $T_{\mathcal{G}_1}, \ldots, T_{\mathcal{G}_q}$ of dumbbell trees. We also show that spanner paths in $G$ can be computed I/O-efficiently. In Section 13.3, we show how to construct a set of $\mathcal{O}(1)$ dumbbell trees for a given WSPD.

## 13.2 The Dumbbell Spanner

Given a set of dumbbell trees for a point set $S \subset \mathbb{R}^d$, as defined in Section 13.1, it is shown in [18] how to derive a $t$-spanner for $S$ from this set of dumbbell trees. We recall this construction in Section 13.2.1 and show that spanner paths in the resulting spanner can be constructed I/O-efficiently. In Section 13.2.2, we discuss a more clever construction of the spanner, also proposed in [18], which guarantees that the spanner has spanner diameter $\mathcal{O}(\log N)$. Given the fact that there exist short spanner paths in the constructed spanner, we show that a simplified version of the algorithm for reporting spanner paths outperforms the algorithm of Section 13.2.1, at least asymptotically.

### 13.2.1 From Tree Paths to Spanner Paths

Let $T_{\mathcal{G}_1}, \ldots, T_{\mathcal{G}_q}$ be the dumbbell trees defined in the previous section. Then the construction of [18] derives spanning trees $G_1, \ldots, G_q$ of $S$ from $T_{\mathcal{G}_1}, \ldots, T_{\mathcal{G}_q}$ and constructs a $t$-spanner $G$ of $S$ which is the union of graphs $G_1, \ldots, G_q$.

Given dumbbell tree $T_{\mathcal{G}_h}$, $1 \leq h \leq q$, the spanning tree $G_h$ of $S$ is obtained as follows: For every node $v \in T_{\mathcal{G}_h}$, we choose a representative point $r(v) \in S$. For a leaf $v$, $r(v) = v$. For an internal node, $r(v) \in \{r(w_1), \ldots, r(w_k)\}$, where $w_1, \ldots, w_k$ are the children of $v$ in $T_{\mathcal{G}_h}$. For every edge $\{v, w\} \in T_{\mathcal{G}_h}$ with $r(v) \neq r(w)$, graph $G_h$ contains an edge $\{r(v), r(w)\}$.

Let $\mathcal{D} = (T, \mathcal{R})$ be the WSPD used to construct trees $T_{\mathcal{G}_1}, \ldots, T_{\mathcal{G}_q}$. Let $p, p' \in S$ be two points, and let $\{A, B\} \in \mathcal{R}$ be the unique pair such that $p \in A$ and $p' \in B$. Let $T_{\mathcal{G}_h}$ be the dumbbell tree containing the dumbbell node $D(A, B)$. Then there is a unique path $P = (p = v_0, v_1, \ldots, v_k = p')$ from $p$ to $p'$ in $T_{\mathcal{G}_h}$. According to the above construction, this path corresponds to a path $\tilde{P} = (p = r(v_0), r(v_1), \ldots, r(v_k) = p')$ in $G_h$. The following lemma shows that path $\tilde{P}$ is a $t$-spanner path.

**Lemma 13.1 (Arya et al. [18, 161])** *Choosing $\delta \leq 1/(3s)$ and $c > 0$ in the construction of dumbbell trees $T_{\mathcal{G}_1}, \ldots, T_{\mathcal{G}_q}$, where $s = \mathcal{O}(d/(t-1))$ is the separation*

*constant of the WSPD used to construct trees* $T_{\mathcal{G}_1}, \ldots, T_{\mathcal{G}_q}$*, the path* $\tilde{P}$ *has Euclidean length* $\ell(\tilde{P}) \leq t \cdot dist_2(p, p')$.

By Lemma 13.1, graph $G$ is a $t$-spanner. Moreover, once tree $T_{\mathcal{G}_h}$ has been identified such that $G_h$ contains the spanner path $\tilde{P}$ as constructed above, path $\tilde{P}$ can easily be reported by traversing the paths from $p$ and $p'$ to their lowest common ancestor in $T_{\mathcal{G}_h}$. Now this is not entirely true because path $P$ may be much longer than $\tilde{P}$, as many nodes along $P$ may have the same representatives. Hence, the edges between these nodes do not contribute new edges to $\tilde{P}$. Observe, however, that all nodes in $T_{\mathcal{G}_h}$ with the same representative $r(v)$ form a path from the leaf $p$ with $p = r(v)$ to some ancestor of $p$ in $T_{\mathcal{G}_h}$. Let $T'_h$ be the tree obtained from $T_{\mathcal{G}_h}$ by compressing all non-leaf nodes on each such path into a single node. Then the path $P'$ in $T'_h$ corresponding to a path $\tilde{P}$ containing $k$ edges contains at most $k + 2$ edges. Hence, tree $T'_h$ can be used to report a $t$-spanner path between $p$ and $p'$.

In order to find tree $T'_h$, we have to find the well-separated pair $\{A, B\}$ such that $p \in A$ and $p' \in B$. Tree $T_{\mathcal{G}_h}$ is the tree containing the dumbbell node $D(A, B)$. Tree $T'_h$ is the compressed version of $T_{\mathcal{G}_h}$. To find pair $\{A, B\}$, we use the following procedure: First we find the lowest common ancestor $A'$ of $p$ and $p'$ in the fair split tree $T$. Then let $T_{A'}$ be the computation tree whose root is the node $(A'', B'')$, where $A''$ and $B''$ are the two children of $A'$ in $T$. (Refer to Section 12.4 on page 251 for a definition of computation trees.) Dumbbell $D(A, B)$ corresponds to a node in $T_{A'}$ and can be found using oblivious search in $T_{A'}$. Hence, we propose the following data structure for reporting spanner paths in $G$. The data structure consists of four parts:

1. A collection of data structures representing trees $T'_1, \ldots, T'_q$. The data structure representing tree $T'_h$ allows the path in $T'_h$ between any two query points $p$ and $p'$ to be reported in $\mathcal{O}(L/(DB))$ I/Os, where $L$ is the number of edges in the reported path.

2. A collection of topology $B$-trees representing the computation trees defined by the given WSPD $\mathcal{D} = (T, \mathcal{R})$. In particular, there is one topology $B$-tree $\mathcal{T}_{A'}$ for

every internal node $A'$ of $T$. Topology $B$-tree $\mathcal{T}_{A'}$ represents the computation tree $T_{A'}$. Every leaf in $\mathcal{T}_{A'}$ representing a leaf $(A, B)$ of $T_{A'}$ is labelled with the index $h$ of tree $T_{\mathcal{G}_h}$ containing dumbbell node $D(A, B)$.

3. A topology $B$-tree $\mathcal{T}$ representing the fair split tree $T$. Every leaf in $\mathcal{T}$ representing an internal node $A'$ of $T$ is labelled with a pointer to the root of topology $B$-tree $\mathcal{T}_{A'}$. Every node $A$ of $T$ is labelled with its number in a preorder numbering of $T$ and the size of subtree $T(A)$ of $T$. Every node $v$ of $\mathcal{T}$ is labelled with the information associated with the root $r_v$ of $T_v$.

4. A table $P$ of pointers to the leaves of trees $T'_1, \ldots, T'_q$. In particular, table $P$ is a $q \times N$ matrix storing for every pair $(h, p)$, $1 \leq h \leq q$, $p \in S$, a pointer to the disk block containing the leaf of tree $T'_h$ which represents point $p$.

**Building the data structure.** First we show that the size of the above data structure is $\mathcal{O}(N)$. It is shown in [100] that there exists a data structure for reporting paths between two leaves of a rooted tree in the number of I/Os stated above. The data structure uses linear space. As there are $\mathcal{O}(1)$ trees $T'_1, \ldots, T'_q$, the data structures for all trees $T'_1, \ldots, T'_q$ use $\mathcal{O}(N)$ space. In [36], it is shown that a topology $B$-tree representing a tree of size $K$ uses $\mathcal{O}(K/B)$ disk blocks. As the total size of all computation trees $T_{A'}$ and fair split tree $T$ is $\mathcal{O}(N)$, the topology trees in the data structure use $\mathcal{O}(N)$ space. Finally, the size of table $P$ is $\mathcal{O}(qN) = \mathcal{O}(N)$.

Next we show that this data structure can be constructed I/O-efficiently. We compute trees $T'_1, \ldots, T'_q$ from trees $T_{\mathcal{G}_1}, \ldots, T_{\mathcal{G}_q}$ as follows: First we process each tree $T_{\mathcal{G}_h}$ from the root toward the leaves to find for every point $p \in S$, the highest internal node $v(p)$ having $p$ as a representative. We apply procedure CopyVertexLabels to replace every edge $\{v, w\}$ in $T_{\mathcal{G}_h}$ with the edge $\{v(r(v)), v(r(w))\}$. A single scan of the resulting edge set is sufficient to remove all loops, which produces the edge set of $T'_h$. This procedure takes $\mathcal{O}(\text{sort}(N))$ I/Os per tree, as it involves one application of time-forward processing and procedure CopyVertexLabels and a scan of the

edge set of $T_{\mathcal{G}_h}$. As there are only $\mathcal{O}(1)$ trees $T_{\mathcal{G}_1}, \ldots, T_{\mathcal{G}_q}$, trees $T'_1, \ldots, T'_q$ can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os.

In [177], it is shown that the data structures representing trees $T'_1, \ldots, T'_q$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. During this construction, the pointers in table $P$ can be generated, but not necessarily in the same order as they are to be stored in $P$. Hence, we represent every pointer by a triple $(p, h, \mathcal{B})$, where $\mathcal{B}$ is the block containing the node of tree $T'_h$ representing point $p \in S$. Sorting these triples lexicographically takes $\mathcal{O}(\text{sort}(qN)) = \mathcal{O}(\text{sort}(N))$ I/Os and produces the final list $P$. The topology $B$-trees representing the fair split tree and the computation trees can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os using a procedure similar to the one presented in Section 12.2.2. (In fact, the procedure is simpler, as no compression of the binary tree is required.) Hence, all parts of the data structure can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os.

**Answering spanner path queries.** To report a spanner path between two query points $p, p' \in S$, we find the lowest common ancestor $A'$ of $p$ and $p'$ in $T$. Then we follow the pointer from $A'$ to the root of the topology tree $\mathcal{T}_{A'}$ representing computation tree $T_{A'}$. We find the leaf $(A, B)$ of tree $\mathcal{T}_{A'}$ so that $p \in A$ and $p' \in B$. This leaf of $\mathcal{T}_{A'}$ stores the index $h$ of the tree $T'_h$ which contains dumbbell $D(A, B)$. Now we query table $P$ to find the two leaves in $T'_h$ representing points $p$ and $p'$. Finally, we use the data structure representing $T'_h$ to report the path in $T'_h$ and hence in $G_h$ from $p$ to $p'$.

Finding the lowest common ancestor $A'$ of $p$ and $p'$ is an oblivious search query in $T$, which can be answered in $\mathcal{O}(\log_B N)$ I/Os using the topology $B$-tree $\mathcal{T}$. To see that this query is oblivious, observe that a subtree $T(v)$ of $T$ contains the LCA of $p$ and $p'$ if and only if $\nu(v) \leq \nu(p), \nu(p') \leq \nu(v) + |T(v)|$, where $\nu(v)$ denotes the preorder number of $v$.

Given that every node $(X, Y)$ of computation tree $T_{A'}$ stores rectangles $R(X)$ and $R(Y)$, locating node $(A, B)$ is an oblivious search query on $T_{A'}$ because subtree $T((X, Y))$ contains node $(A, B)$ if and only if $p \in X$ and $p' \in Y$. Hence, node $(A, B)$ can be found in $\mathcal{O}(\log_B N)$ I/Os using topology $B$-tree $\mathcal{T}_{A'}$.

Querying table $P$ takes two I/Os, one per point, and reporting the path in $T_i'$ from $p$ to $p'$ takes $\mathcal{O}(L/(DB))$ I/Os. Hence, a spanner path query can be answered in $\mathcal{O}(\log_B N + L/(DB))$ I/Os.

By Theorem 13.3, the set $T_{\mathcal{G}_1}, \ldots, T_{\mathcal{G}_q}$ of dumbbell trees can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os. Hence, the above discussion leads to the following result.

**Theorem 13.1** *Let $S$ be a set of $N$ points in $\mathbb{R}^d$. There exists a $t$-spanner $G$ of linear size for $S$ which can be represented by a data structure of linear size that allows spanner paths between two query points $p, p' \in S$ to be reported in $\mathcal{O}(\log_B N + L/(DB))$ I/Os, where $L$ is the length of the reported path. The spanner and the data structure can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space.*

### 13.2.2 A Spanner of Logarithmic Diameter

By choosing representatives of the nodes in each tree $T_{\mathcal{G}_h}$ more carefully in the construction presented in the previous section, a spanner of spanner diameter $\mathcal{O}(\log N)$ can be obtained: For every node $v \in T_{\mathcal{G}_h}$, let $\omega(v)$ be the number of leaves that are descendants of $v$ in $T_{\mathcal{G}_h}$. As before, if $v$ is a leaf, then $r(v) = v$. If $v$ is an internal node with children $w_1, \ldots, w_k$, let $r(v) = r(v')$ where $v' \in \{w_1, \ldots, w_k\}$ and $\omega(v') = \max\{\omega(w_j) : 1 \leq j \leq k\}$.

**Lemma 13.2** *Let $T_{\mathcal{G}_h}$ be any of the dumbbell trees, $p$ and $p'$ be two query points. Then the path $\tilde{P}$ from $p$ to $p'$ in $G_h$ corresponding to the path $P$ from $p$ to $p'$ in $T_{\mathcal{G}_h}$ contains $\mathcal{O}(\log N)$ edges.*

*Proof.* To prove the lemma it is sufficient to show that along any leaf-to-root path $P$ in $T = T_{\mathcal{G}_h}$, there are at most $\log N$ different representatives. So consider a node $v$ so that $r(v) \neq r(p(v))$. Then $\omega(p(v)) \geq 2\omega(v)$ because $r(p(v)) = r(v')$, for some sibling $v'$ of $v$ with $\omega(v') \geq \omega(v)$, and $\omega(p(v)) = \omega(v) + \omega(v')$. Hence, if there were $k > \log N$ nodes $v_1, \ldots, v_k$ on path $P$ so that $r(v_i) \neq r(p(v_i))$, we would obtain by induction that $\omega(p(v_k)) > N$, which leads to a contradiction. □

Lemma 13.2 and Theorem 13.1 together imply that a spanner path in $G$ can be reported in $\mathcal{O}(\log_B N + \log_2 N/(DB))$ I/Os. However, there are only $\mathcal{O}(1)$ dumbbell trees $T'_1, \ldots, T'_q$. Hence, the following alternative procedure finds a spanner path in $\mathcal{O}(\log_2 N/(DB))$ I/Os: We find the paths $\tilde{P}_1, \ldots, \tilde{P}_q$ between $p$ and $p'$ in graphs $G_1, \ldots, G_q$ by querying all trees $T'_1, \ldots, T'_q$ without knowing which of these trees contains dumbbell $D(A, B)$. Then we report the shortest of paths $\tilde{P}_1, \ldots, \tilde{P}_q$.

**Theorem 13.2** *Let $S$ be a set of $N$ points in $\mathbb{R}^d$. There exists a $t$-spanner $G$ of linear size and spanner diameter $\mathcal{O}(\log N)$ for $S$ which can be represented by a data structure of linear size that allows a spanner path between two query points $p, p' \in S$ to be reported in $\mathcal{O}(\log N/(DB))$ I/Os. The spanner and the data structure can be constructed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os and linear space.*

## 13.3 Constructing the Dumbbell Trees

Motivated by the fact that dumbbell trees can be used to derive $t$-spanners which allow spanner paths to be reported I/O-efficiently, we show in the rest of this chapter how to compute such a collection of $\mathcal{O}(1)$ dumbbell trees $T_{\mathcal{G}_1}, \ldots, T_{\mathcal{G}_q}$ I/O-efficiently.

In Sections 13.3.1 through 13.3.3, we show how to partition the set of dumbbells defined by a WSPD $\mathcal{D} = (T, \mathcal{R})$ into $\mathcal{O}(1)$ groups $\mathcal{G}_1, \ldots, \mathcal{G}_q$ with the length grouping, empty region, and nesting properties. In Section 13.3.4, we present an algorithm to compute dumbbell trees $T_{\mathcal{G}_1}, \ldots, T_{\mathcal{G}_q}$ from these groups.

### 13.3.1 The Length Grouping Property

The idea of the length grouping algorithm can be described as follows [161]: Let $\beta = \delta/2$. Then the dumbbells are partitioned into an infinite number of groups $\mathcal{G}'_0, \mathcal{G}'_1, \ldots$ so that the lengths of the dumbbells in group $\mathcal{G}'_i$ lie in the interval $(\beta^{i+1}L, \beta^i L]$, where $L$ is the length of the longest dumbbell induced by $\mathcal{R}$. Observe that only a finite number of these groups are non-empty. Each group $\mathcal{G}'_i$ is further divided into groups $\mathcal{G}'_{i,j}$, $0 \leq j < \lceil -\log \beta \rceil$, where the lengths of the dumbbells in group $\mathcal{G}'_{i,j}$ lie in the

**Procedure** LengthGrouping

**Input:** A point set $S \subset \mathbb{R}^d$ and a WSPD $\mathcal{D} = (T, \mathcal{R})$ of $S$.
**Output:** A partition of the dumbbells induced by $\mathcal{R}$ into groups $\mathcal{G}''_0, \ldots, \mathcal{G}''_r$, $r = \lceil -\log \beta \rceil$, so that each group $\mathcal{G}''_j$ has the length grouping property.

1: Sort the set of dumbbells by their lengths.
2: $l \leftarrow$ the length of the longest dumbbell in $\mathcal{D}$.
3: {The following while-loop scans the sorted list of dumbbells.}
4: **while** not all dumbbells have been read **do**
5: Partition interval $(\beta l, l]$ into subintervals $(\beta l, 2\beta l], (2\beta l, 4\beta l], \ldots$.
6: **while** $\ell(D(A,B)) > \beta l$ for the next dumbbell $D(A,B)$ to be read **do**
7: Read dumbbell $D(A,B)$
8: Assign a label $\gamma(D(A,B)) = (j,l)$ to dumbbell $D(A,B)$, where $\ell(D(A,B)) \in (2^j \beta l, 2^{j+1} \beta l]$.
9: **end while**
10: Let $l$ be the length of the next dumbbell to be read.
11: **end while**
12: Sort the set of dumbbells lexicographically by their labels $\gamma(D(A,B))$.

**Algorithm 13.1**
Partitioning the dumbbells induced by a WSPD into groups having the length grouping property.

interval $(2^j \beta^{i+1} L, \min\{2^{j+1}\beta^{i+1}L, \beta^i L\}]$. Now let $\mathcal{G}''_j = \bigcup_{i=0}^{\infty} \mathcal{G}'_{i,j}$. If two dumbbells in $\mathcal{G}''_j$ come from the same group $\mathcal{G}'_{i,j}$, their lengths differ by a factor of at most two. Otherwise, their lengths differ by a factor of at least $1/(2\beta) = 1/\delta$, as required.

We apply Algorithm 13.1 to compute groups $\mathcal{G}''_0, \ldots, \mathcal{G}''_r$, $r = \lceil -\log \beta \rceil$. The algorithm may produce groups that differ from the partition defined above because it may choose empty intervals in the list of dumbbell lengths to span more than an interval $(\beta^{i+1} L, \beta^i L]$; but the length grouping property is preserved. The dumbbells in each group are stored consecutively, and each group $\mathcal{G}''_j$ is partitioned into its constituent subgroups $\mathcal{G}'_{i,j}$. While the latter partition is not required at this point, it will be used by Algorithm 13.2, which ensures the empty region property.

Procedure LengthGrouping takes $\mathcal{O}(\text{sort}(N))$ I/Os: Lines 1 and 12 sort lists of size $\mathcal{O}(N)$. The loop in Lines 4–11 can be realized in a single scan of the sorted

list of dumbbells. We have to show that the groups produced by this procedure have the length-grouping property.

Consider two dumbbells in group $\mathcal{G}''_j$. If their lengths fall into the same interval $(\beta l, l]$, their lengths differ by a factor of at most two. Otherwise, the length of the shorter dumbbell lies in an interval $(2^j \beta l', 2^{j+1} l']$ and the length of the longer dumbbell lies in an interval $(2^j \beta l, 2^{j+1} l]$, where $l' \leq \beta l$. Hence, the lengths of these two dumbbells differ by a factor of at least $1/(2\beta) = 1/\delta$.

**Lemma 13.3** *Algorithm 13.1 takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os and linear space to compute a partition of the dumbbells induced by a WSPD of a set $S$ of $N$ points in $\mathbb{R}^d$ into $\mathcal{O}(1)$ groups so that each group has the length grouping property.*

### 13.3.2 The Empty Region Property

Given the partition of the set of dumbbells induced by $\mathcal{D}$ into $\mathcal{O}(1)$ groups $\mathcal{G}''_0, \ldots, \mathcal{G}''_r$, as computed by Algorithm 13.1, Algorithm 13.2 partitions each group $\mathcal{G}''_j$ into a constant number of subgroups $\mathcal{G}''_{j,1}, \ldots, \mathcal{G}''_{j,s}$, each having the empty region property. Each group $\mathcal{G}''_{j,k}$ also has the length grouping property because $\mathcal{G}''_j$ has this property. Since there are $\mathcal{O}(1)$ groups $\mathcal{G}''_0, \ldots, \mathcal{G}''_r$ and each of them is partitioned into $\mathcal{O}(1)$ subgroups $\mathcal{G}''_{j,1}, \ldots, \mathcal{G}''_{j,s}$, the set of dumbbells induced by $\mathcal{D}$ is partitioned into $\mathcal{O}(1)$ groups, each having the length grouping and empty region properties.

Assuming that graph $P_{i,j}$ can be colored with $\mathcal{O}(1)$ colors as attempted in Line 4 of Algorithm 13.2, the correctness of the algorithm is easy to see. In particular, no two nodes in $P_{i,j}$ with the same color are adjacent, so that it follows from the definition of $P_{i,j}$ that the dumbbells in each group $\mathcal{G}'_{i,j,k}$ have the empty region property. This implies that groups $\mathcal{G}''_{j,1}, \ldots, \mathcal{G}''_{j,s}$ have the empty region property because two dumbbells in the same group $\mathcal{G}''_{j,k}$ come either from the same group $\mathcal{G}'_{i,j,k}$ or differ by more than two in length.

In the remainder of this subsection, we show that graph $P_{i,j}$ has bounded degree (Lemma 13.8) and can be constructed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os (Lemma 13.9). Thus, we obtain the following lemma.

**Procedure** EMPTYREGION

**Input:** A group of dumbbells $\mathcal{G}''_j$ having the length grouping property.
**Output:** A partition of $\mathcal{G}''_j$ into subgroups $\mathcal{G}''_{j,1}, \ldots, \mathcal{G}''_{j,s}$ having the empty region property.

1: Let $\mathcal{G}'_{1,j}, \ldots, \mathcal{G}'_{t,j}$ be the subgroups of $\mathcal{G}''_j$ so that the dumbbells in each such group differ in length by a factor of at most two.
2: **for** $1 \leq i \leq t$ **do**
3: Construct a proximity graph $P_{i,j}$ of group $\mathcal{G}'_{i,j}$:
The nodes of $P_{i,j}$ represent the dumbbells in $\mathcal{G}'_{i,j}$.
There is an edge between two nodes $D(A,B)$ and $D(A',B')$ in graph $P_{i,j}$ if $\text{dist}_2(D(A,B), D(A',B')) < c \cdot \min(\ell(D(A,B)), \ell(D(A',B')))$.
4: Color graph $P_{i,j}$ with $s = \mathcal{O}(1)$ colors.
5: For $1 \leq k \leq s$, let $\mathcal{G}'_{i,j,k}$ be the set of dumbbells whose corresponding nodes in $P_{i,j}$ have color $k$.
6: **end for**
7: Form groups $\mathcal{G}''_{j,1}, \ldots, \mathcal{G}''_{j,s}$ as $\mathcal{G}''_{j,k} = \bigcup_{i=1}^{t} \mathcal{G}'_{i,j,k}$.

**Algorithm 13.2**
Partitioning a set of dumbbells into groups having the empty region property.

**Lemma 13.4** *Given a group $\mathcal{G}''_j$ of dumbbells having the length grouping property, Algorithm 13.2 takes $\mathcal{O}(\text{sort}(N))$ I/Os to partition group $\mathcal{G}''_j$ into $\mathcal{O}(1)$ subgroups $\mathcal{G}''_{j,1}, \ldots, \mathcal{G}''_{j,s}$, each having the length grouping and empty region properties.*

*Proof.* The correctness of procedure EMPTYREGION follows from the above discussion and Lemma 13.8. Line 1 of the algorithm does not require any computation, as the final sorting step of Algorithm 13.1 leaves the dumbbells in $\mathcal{G}''_j$ partitioned into groups $\mathcal{G}'_{1,j}, \ldots, \mathcal{G}'_{t,j}$. By Lemma 13.9, proximity graphs $P_{1,j}, \ldots, P_{t,j}$ can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os. Each proximity graph $P_{i,j}$ has bounded degree and size $\mathcal{O}(|\mathcal{G}'_{i,j}|)$, by Lemma 13.8. Hence, by Lemma 6.1, it can be colored with $\mathcal{O}(1)$ colors in $\mathcal{O}(\text{sort}(|\mathcal{G}_{i,j}|))$ I/Os. Since, $\sum_{i=1}^{t} |\mathcal{G}'_{i,j}| = \mathcal{O}(N)$, coloring all graphs $P_{1,j}, \ldots, P_{t,j}$ takes $\mathcal{O}(\text{sort}(N))$ I/Os. Once every dumbbell has been assigned a color, the construction of groups $\mathcal{G}''_{j,1}, \ldots, \mathcal{G}''_{j,s}$ can be achieved by sorting the dumbbells in $\mathcal{G}''_j$ lexicographically by labels $\gamma'(D(A,B)) = (j,k,l)$, where $\gamma(D(A,B)) = (j,l)$ is the label

assigned to dumbbell $D(A, B)$ by Algorithm 13.1, and $k$ is the color assigned to dumbbell $D(A, B)$. This takes $\mathcal{O}(\text{sort}(|\mathcal{G}''_j|)) = \mathcal{O}(\text{sort}(N))$ I/Os. Hence, Algorithm 13.2 takes $\mathcal{O}(\text{sort}(N))$ I/Os. □

Note that Algorithm 13.2 leaves the dumbbells in each group $\mathcal{G}''_{j,k}$ partitioned into groups $\mathcal{G}'_{i,j,k}, \ldots, \mathcal{G}'_{l,j,k}$. This partition is used by Algorithm 13.3, which constructs a dumbbell tree for each group $\mathcal{G}''_{j,k}$.

To complete the proof of Lemma 13.4, we have to show that graph $P_{i,j}$ has bounded degree, for every group $\mathcal{G}'_{i,j}$, and that it can be computed in $\mathcal{O}(\text{sort}(|\mathcal{G}'_{i,j}|))$ I/Os.

#### 13.3.2.1 Bounding the Degree of the Proximity Graph

The bound on the degree of graph $P_{i,j}$ is obtained using packing arguments: We prove that for any dumbbell $D(A, B) \in \mathcal{G}'_{i,j}$, there are only $\mathcal{O}(1)$ nodes in the fair split tree whose bounding rectangles can be heads of dumbbells in $\mathcal{G}'_{i,j}$ that are too close to $D(A, B)$. To prove this fact, we show that the number of such rectangles of minimal size is constant and that each such minimal rectangle has only $\mathcal{O}(1)$ ancestors which may be heads of dumbbells in $\mathcal{G}'_{i,j}$. Given this bound on the number of dumbbell heads that are too close to $D(A, B)$, it is now sufficient to bound the number of dumbbells $D(A', B') \in \mathcal{G}'_{i,j}$, for any given node $A' \in T$, by a constant. We show these two facts in Lemma 13.8. We use the next three lemmas to bound the number of nodes on a root-to-leaf path in $T$ whose bounding rectangles can be heads of dumbbells in $\mathcal{G}'_{i,j}$.

**Lemma 13.5** *Let $D(A, B)$ be a dumbbell whose length is $l$. Then $\ell_{\max}(p(A)) > \frac{2l}{(s+4)\sqrt{d}}$ and $\ell_{\max}(p(B)) > \frac{2l}{(s+4)\sqrt{d}}$.*

*Proof.* Let $A' = p(A)$ and $B' = p(B)$, and assume w.l.o.g. that $B' \prec A'$. Then the parent of node $(A, B)$ in the computation tree containing node $(A, B)$ is node $(A, B')$. Let $(A', B'')$ be the lowest ancestor of node $(A, B)$ in the computation tree so that the well-separated pair $\{A', B''\}$ does not contain set $A$. We show the lemma for $A'$. The argument for $B'$ is similar.

Since node $(A', B'')$ has a child $(A, B'')$ in the computation tree, $\ell_{\max}(A') \geq \ell_{\max}(B'')$ and sets $A'$ and $B''$ are not well-separated. The former implies that both rectangles $R(A')$ and $R(B'')$ can be enclosed in two balls of radius $r = \frac{\sqrt{d}}{2}\ell_{\max}(A')$. The latter implies that any two such balls enclosing $R(A')$ and $R(B'')$ have distance less than $sr$. Since $R(A)$ and $R(B)$ are contained in $R(A')$ and $R(B'')$, respectively, this implies that $l < (s+4)r$, so that $\ell_{\max}(A') = \frac{2r}{\sqrt{d}} > \frac{2l}{(s+4)\sqrt{d}}$. □

**Lemma 13.6** *Let $D(A, B)$ be a dumbbell whose length is $l$. Then $\ell_{\min}(\tilde{R}(A)) > \frac{2l}{(3s+12)\sqrt{d}}$ and $\ell_{\min}(\tilde{R}(B)) > \frac{2l}{(3s+12)\sqrt{d}}$.*

*Proof.* We prove the lemma for rectangle $\tilde{R}(A)$. The proof for rectangle $\tilde{R}(B)$ is similar. By Lemma 13.5, $\ell_{\max}(p(A)) > \frac{2l}{(s+4)\sqrt{d}}$, which implies that $\ell_{\max}(\hat{R}(p(A))) > \frac{2l}{(s+4)\sqrt{d}}$. If $i_{\min}(\tilde{R}(A)) = i_{\max}(\hat{R}(p(A)))$, then $\ell_{\min}(\tilde{R}(A)) > \frac{2l}{(3s+12)\sqrt{d}}$ because the split of rectangle $\hat{R}(p(A))$ is fair. Otherwise, $\ell_{\min}(\tilde{R}(A)) = \ell_{\min}(\hat{R}(p(A))) > \frac{2l}{(3s+12)\sqrt{d}}$ because rectangle $\hat{R}(p(A))$ is a box. □

Next we show that only a constant number of nodes along any root-to-leaf path in $T$ can be heads of dumbbells whose lengths lie in an interval $(l, 2l]$.

**Lemma 13.7** *Let $D(A, B)$ and $D(A', B')$ be two dumbbells whose lengths lie in the interval $(l, 2l]$ and so that $A'$ is an ancestor of $A$ in the fair split tree. Then $A' = p^{(k)}(A)$, where $k = \mathcal{O}(d \log d)$.*

*Proof.* Assume that $k > d$ because otherwise the lemma holds. By Lemma 13.5, $\ell_{\max}(\hat{R}(p(A))) \geq \frac{2l}{(s+4)\sqrt{d}}$. By Lemma 12.6, $\ell_{\max}(\hat{R}(A'')) \leq \frac{12l}{s+2}$, where $A''$ is the descendant of $A'$ so that $A' = p^{(d)}(A'')$. Since $k > d$, node $A''$ exists and is an ancestor of $p(A)$. By Lemma 12.7, $A'' = p^{(k')}(p(A))$, where $k' \leq d \log_{3/2} \frac{6\sqrt{d}(s+4)}{s+2} \leq d \log_{3/2} \left(12\sqrt{d}\right)$, so that $k \leq 1 + d\left(1 + \log_{3/2}\left(12\sqrt{d}\right)\right) = \mathcal{O}(d \log d)$. □

We are now ready to prove that the proximity graph $P$ of a group $\mathcal{G}'_{i,j}$ has constant degree. In the following lemma, $c$ is the constant used in the definition of the empty region property on page 262.

**Lemma 13.8** *The degree of the proximity graph $P_{i,j}$ of a group $\mathcal{G}'_{i,j}$ is bounded by $\mathcal{O}\left((s+1)\sqrt{(c+1)d}\right)^{2d}$.*

*Proof.* Let $D(A,B)$ be a dumbbell in group $\mathcal{G}'_{i,j}$, and let all dumbbells in $\mathcal{G}'_{i,j}$ have lengths in the interval $(l,2l]$. Then the lemma follows from the following two claims, which we prove below:

*(i) There are $\mathcal{O}\left((c+1)(s+1)\sqrt{d}\right)^d$ nodes $A'$ in $T$ so that there is a dumbbell $D(A',B') \in \mathcal{G}'_{i,j}$ and $\mathrm{dist}_2(R(A'), D(A,B)) \leq 2cl$.*

*(ii) For any node $A \in T$, there are $\mathcal{O}\left((s+1)\sqrt{d}\right)^d$ dumbbells in $\mathcal{G}'_{i,j}$ which have rectangle $R(A)$ as a head.*

(i) Let $R$ be a rectangle of minimum size around $D(A,B)$ so that for all $p \notin R$, $\mathrm{dist}_2(p, D(A,B)) > 2cl$. Since $D(A,B) \in \mathcal{G}'_{i,j}$, $\ell(D(A,B)) \leq 2l$, $\ell_{\max}(A) \leq 4l/s$, and $\ell_{\max}(B) \leq 4l/s$. Hence, $\ell_{\max}(R) \leq 2l + 4l/s + 4cl = 2l(1+2c+2/s)$.

It follows from the definition of rectangle $R$ that every dumbbell head $R(A')$ so that $\mathrm{dist}_2(R(A'), D(A,B)) \leq 2cl$ must intersect $R$. Since $R(A') \subseteq \tilde{R}(A')$, this implies that $\tilde{R}(A') \cap R \neq \emptyset$. As we are interested only in heads $R(A')$ of dumbbells $D(A',B') \in \mathcal{G}'_{i,j}$, Lemma 13.6 implies that $\ell_{\min}(\tilde{R}(A')) \geq \frac{2l}{(3s+12)\sqrt{d}}$ because dumbbell $D(A',B')$ has length at least $l$.

Now let $A_1, \ldots, A_t$ be the nodes in $T$ so that the bounding box $R(A_x)$ of each node $A_x$, $1 \leq x \leq t$, is a head of a dumbbell in $\mathcal{G}'_{i,j}$, $\mathrm{dist}_2(R(A_x), D(A,B)) \leq 2cl$, and none of the descendants of $A_x$ has this property. Then the split rectangles $\tilde{R}(A_1), \ldots, \tilde{R}(A_t)$ are disjoint, $\ell_{\min}(\tilde{R}(A_x)) \geq \frac{2l}{(3s+12)\sqrt{d}}$, and $\tilde{R}(A_x) \cap R \neq \emptyset$, for all $1 \leq x \leq t$. This implies that

$$\begin{aligned} t &\leq \left( \left\lfloor \frac{\ell_{\max}(R)}{\min_{1\leq x\leq t}(\ell_{\min}(\tilde{R}(A_x)))} \right\rfloor + 1 \right)^d \\ &\leq \left( (1+2c+2/s)(3s+12)\sqrt{d} + 1 \right)^d \\ &= \mathcal{O}\left( (c+1)(s+1)\sqrt{d} \right)^d . \end{aligned}$$

Now it remains to observe that every head $R(A')$ of a dumbbell $D(A', B') \in \mathcal{G}'_{i,j}$ with $\text{dist}_2(R(A'), D(A, B)) \leq 2cl$ is an ancestor of some node $A_x$ in $T$, $1 \leq x \leq t$. By Lemma 13.7, the number of these ancestors of node $A_x$ is bounded by $\mathcal{O}(d \log d)$. Hence, the total number of dumbbell heads $R(A')$ with $\text{dist}_2(R(A'), D(A, B)) \leq 2cl$ and $D(A', B') \in \mathcal{G}'_{i,j}$ is $\mathcal{O}\left((c+1)(s+1)\sqrt{d}\right)^d \cdot \mathcal{O}(d \log d) = \mathcal{O}\left((c+1)(s+1)\sqrt{d}\right)^d$.

(ii) To prove the second claim, we fix some dumbbell head $R(A)$. Let $B_1, \ldots, B_t$ be the nodes in $T$ so that for all $1 \leq x \leq t$, there is a dumbbell $D(A, B_x) \in \mathcal{G}'_{i,j}$. Then the split rectangles $\tilde{R}(B_1), \ldots, \tilde{R}(B_t)$ are disjoint, have minimum side length $\ell_{\min}(\tilde{R}(B_x)) \geq \frac{2l}{(3s+12)\sqrt{d}}$, and intersect a cube $R$ with the same center as $R(A)$ and side length $4l$. Hence, $t \leq \left((6s+24)\sqrt{d}+1\right)^d = \mathcal{O}\left((s+1)\sqrt{d}\right)^d$. □

For fixed parameters $c$, $s$, and $d$, Lemma 13.8 shows that the degree of the proximity graph $P_{i,j}$ of any group $\mathcal{G}'_{i,j}$ is constant, which establishes the correctness of Algorithm 13.2.

#### 13.3.2.2 Computing the Proximity Graph

In the remainder of this section, we describe an algorithm for constructing proximity graphs $P_{1,j}, \ldots, P_{t,j}$ in $\mathcal{O}(\text{sort}(N))$ I/Os. Instead of constructing each graph separately, we construct a graph $P_j = P_{1,j} \cup \cdots \cup P_{t,j}$. This is necessary because otherwise we would need an algorithm that constructs each graph $P_{i,j}$ in $o(\text{sort}(N))$ I/Os or a bound of $t = \mathcal{O}(1)$ on the number of groups $\mathcal{G}'_{1,j}, \ldots, \mathcal{G}'_{t,j}$, in order to obtain a total I/O-complexity of $\mathcal{O}(\text{sort}(N))$ for the construction of proximity graphs $P_{1,j}, \ldots, P_{t,j}$. We do not know how to achieve either.

The vertex set of $P_j$ is easily constructed in $\mathcal{O}(\text{scan}(|\mathcal{G}''_j|))$ I/Os, as $P_j$ contains one vertex per dumbbell in $\mathcal{G}''_j$. To construct the edge set of $P_j$, we create a set $Q(A, B)$ of restricted containment queries on the fair split tree $T$, for every dumbbell $D(A, B) \in \mathcal{G}''_j$. Let $D(A, B) \in \mathcal{G}'_{i,j}$. Then the construction of set $Q(A, B)$ ensures that for a dumbbell $D(A', B') \in \mathcal{G}'_{i,j}$ which has distance at most $2cl$ from $D(A, B)$, the set $H(A, B)$ of answers to the queries in $Q(A, B)$ contains at least one of the nodes $A'$ or $B'$. Below we show that for every dumbbell $D(A, B) \in \mathcal{G}''_j$, $|Q(A, B)| = \mathcal{O}(1)$ and

$|H(A,B)| = \mathcal{O}(1)$. This allows us to show the following lemma, which establishes the I/O-bound of Algorithm 13.2.

**Lemma 13.9** *Given the set of dumbbells in group $\mathcal{G}''_j$, the proximity graph $P_j$ of group $\mathcal{G}''_j$ can be constructed in $\mathcal{O}(\text{sort}(\mathcal{G}''_j))$ I/Os.*

*Proof.* As argued above, the vertex set of $P_j$ can be constructed in $\mathcal{O}(\text{scan}(|\mathcal{G}''_j|))$ I/Os. By Lemma 13.11 below, sets $Q(A,B)$ and $H(A,B)$ have constant size, for every dumbbell $D(A,B) \in \mathcal{G}''_j$. Hence, the restricted containment queries in all sets $Q(A,B)$ can be answered in $\mathcal{O}(\text{sort}(|\mathcal{G}''_j|))$ I/Os, by Lemma 12.9.

Given the sets $H(A,B)$ of answers to containment queries in sets $Q(A,B)$, we construct the edge set of $P_j$ as follows: For each group $\mathcal{G}'_{i,j}$, we scan the answer sets $H(A,B)$, for all dumbbells $D(A,B) \in \mathcal{G}'_{i,j}$, and create a list $L_i$ containing a triple $(A', D(A,B))$, for every dumbbell $D(A,B) \in \mathcal{G}'_{i,j}$ and every head $A' \in H(A,B)$. We sort list $L_i$ lexicographically and the dumbbells in $\mathcal{G}'_{i,j}$ by their first heads. Then we scan the sorted list $\mathcal{G}'_{i,j}$ and list $L_i$ to add for every triple $(A', D(A,B)) \in L_i$ and every dumbbell $D(A',B') \in \mathcal{G}'_{i,j}$ whose first head is $A'$ an edge $(D(A,B), D(A',B'))$ to $P_j$. We sort the dumbbells in $\mathcal{G}'_{i,j}$ by their second heads and repeat the scan to generate an edge $(D(A,B), D(A',B'))$ for every triple $(B', D(A,B))$ and every dumbbell $D(A',B')$ whose second head is $B'$. This procedure may generate every edge in $P_j$ more than once because for a pair of dumbbells $D(A,B)$ and $D(A',B')$, both nodes $A$ and $B$ may be in $H(A',B')$ and both nodes $A'$ and $B'$ may be in $H(A,B)$. Also, the same head may be reported for a number of query points in $Q(A,B)$ and $Q(A',B')$. We apply procedure DuplicateRemoval to the edge set of $P_j$ to obtain the final graph $P_j$.

By Lemma 13.11, the total size of all answer sets $H(A,B)$, $D(A,B) \in \mathcal{G}'_{i,j}$, is $\mathcal{O}(|\mathcal{G}'_{i,j}|)$. Hence, list $L_i$ has size $\mathcal{O}(|\mathcal{G}'_{i,j}|)$, so that the above procedure sorts and scans lists of size $\mathcal{O}(|\mathcal{G}'_{i,j}|)$ and applies procedure DuplicateRemoval to a list of this size. Hence, the construction of the edge set of $P_j$ takes $\mathcal{O}(\text{sort}(|\mathcal{G}'_{i,j}|))$ I/Os per subgroup $\mathcal{G}'_{i,j}$ of $\mathcal{G}''_j$, $\mathcal{O}(\text{sort}(|\mathcal{G}''_j|))$ I/Os in total. □

It remains to define the query set $Q(A,B)$, for every dumbbell $D(A,B) \in \mathcal{G}'_{i,j}$, and bound its size. Let the dumbbells in $\mathcal{G}'_{i,j}$ have lengths in the interval $(l, 2l]$. Then let $R(A,B)$ be a rectangle of minimum size around $D(A,B)$ so that for every point $p \notin R(A,B)$, $\text{dist}_2(p, D(A,B)) > 2cl + \frac{2l}{(3s+12)\sqrt{d}}$. We partition $R(A,B)$ into a regular grid whose cells have side length $\frac{2l}{(3s+12)\sqrt{d}}$. For every grid point $p$, we add a query $\left(p, \frac{4l}{s}, \frac{2l}{(3s+12)\sqrt{d}}\right)$ to $Q(A,B)$. The following lemma shows that every dumbbell $D(A',B') \in \mathcal{G}'_{i,j}$ which is too close to $D(A,B)$ has at least one head matching a query in set $Q(A,B)$.

**Lemma 13.10** *Let $D(A,B)$ and $D(A',B')$ be two dumbbells in group $\mathcal{G}'_{i,j}$ so that $dist_2(D(A,B), D(A',B')) \leq 2cl$. Then there exists a query $\left(p, \frac{4l}{s}, \frac{2l}{(3s+12)\sqrt{d}}\right)$ in set $Q(A,B)$ which reports node $A'$ or $B'$.*

*Proof.* Assume w.l.o.g. that $\text{dist}_2(R(A'), D(A,B)) \leq 2cl$. Since $R(A') \subseteq \tilde{R}(A')$, this implies that $\text{dist}_2(\tilde{R}(A'), D(A,B)) \leq 2cl$, so that rectangle $R' = R(A,B) \cap \tilde{R}(A')$ has minimal side length $\ell_{\min}(R') \geq \frac{2l}{(3s+12)\sqrt{d}}$. Hence, rectangle $R' \subseteq \tilde{R}(A')$ contains at least one grid point $p$. Moreover, $\ell_{\max}(A') \leq \frac{4l}{s}$ and $\ell_{\min}(\tilde{R}(A')) \geq \frac{2l}{(3s+12)\sqrt{d}}$, by Lemma 13.6. Hence, node $A'$ is reported for query $\left(p, \frac{4l}{s}, \frac{2l}{(3s+12)\sqrt{d}}\right) \in Q(A,B)$. □

The next lemma bounds the size of sets $Q(A,B)$ and $H(A,B)$, for any dumbbell $D(A,B) \in \mathcal{G}''_j$, and thereby completes the proof of Lemma 13.9.

**Lemma 13.11** *For fixed parameters, $c$, $s$, and $d$, $|Q(A,B)| = \mathcal{O}(1)$ and $|H(A,B)| = \mathcal{O}(1)$.*

*Proof.* The maximal side length $\ell_{\max}(R(A,B))$ of rectangle $R(A,B)$ is at most $2l\left(1 + 2c + \frac{2}{s} + \frac{2}{(3s+12)\sqrt{d}}\right) \leq 2l(2c+5)$. Since every grid cell has side length $\frac{2l}{(3s+12)\sqrt{d}}$, there are at most $\left((2c+5)(3s+12)\sqrt{d}\right)^d = \mathcal{O}\left((c+1)(s+1)\sqrt{d}\right)^d$ grid cells. Since every grid cell has $2^d$ vertices, the number of grid points is $\mathcal{O}\left((c+1)(s+1)\sqrt{d}\right)^d = \mathcal{O}(1)$. Every grid point gives rise to one query in $Q(A,B)$. Hence, $|Q(A,B)| = \mathcal{O}(1)$. By Lemma 12.8, the number of answers for every query $\left(p, \frac{4l}{s}, \frac{2l}{(3s+12)\sqrt{d}}\right) \in Q(A,B)$,

is $\mathcal{O}\left(d \log\left(2(3s+12)\sqrt{d}/s\right)\right) = \mathcal{O}(1)$, so that $|H(A,B)| = \mathcal{O}(|Q(A,B)|) = \mathcal{O}(1)$. □

### 13.3.3 The Nesting Property

The groups of dumbbells constructed in the previous two sections have the length grouping and empty region properties; but it is not obvious that they should have the nesting property. It is easy to see that for two dumbbells $D(A,B)$ and $D(A',B')$, their heads are either completely disjoint, or w.l.o.g. $R(A) \subseteq R(A')$ because the dumbbell heads are nodes of the fair split tree $T$. However, there is no reason to believe that $D(A',B')$ is necessarily the longer of the two dumbbells. The following lemma shows that this property can be guaranteed if parameter $\delta$ in the length grouping property is chosen small enough.

**Lemma 13.12** *Let $D(A,B)$ and $D(A',B')$ be two dumbbells in the same group $\mathcal{G}''_{j,k}$ so that $\ell(D(A,B)) \leq \delta\ell(D(A',B'))$, and let $\delta < \frac{1}{2\sqrt{d}}$. Then either $D(A,B)$ and $D(A',B')$ are disjoint, or w.l.o.g. node $A$ is a descendant of node $A'$.*

*Proof.* By Lemma 13.5, $\ell_{\max}(p(A')) \geq \frac{2\ell(D(A',B'))}{(s+4)\sqrt{d}}$. Moreover, $\ell_{\max}(A) \leq \frac{2\ell(D(A,B))}{s+2} \leq \frac{2\delta\ell(D(A',B'))}{s+2}$ because the pair $\{A,B\}$ is well-separated. Hence, $\frac{\ell_{\max}(A)}{\ell_{\max}(p(A'))} \leq \frac{\delta(s+4)\sqrt{d}}{s+2} \leq 2\delta\sqrt{d} < 1$. That is, $\ell_{\max}(A) < \ell_{\max}(p(A'))$. This implies that either $R(A)$ and $R(p(A'))$ are disjoint or $A$ is a proper descendant of $p(A')$. The latter implies that $A$ is a descendant of $A'$. □

### 13.3.4 Constructing the Dumbbell Trees

By Lemmas 13.3, 13.4, and 13.12, a partition of the set of dumbbells induced by a WSPD $\mathcal{D}$ of a point set $S$ into $\mathcal{O}(1)$ groups, each having the length grouping, empty region, and nesting properties can be obtained in $\mathcal{O}(\text{sort}(N))$ I/Os. In this section, we provide an algorithm to construct a dumbbell tree $T_{\mathcal{G}''_{j,k}}$, for each such group $\mathcal{G}''_{j,k}$. Algorithm 13.3 provides the details.

**Procedure** BUILDDUMBBELLTREE

**Input:** A fair split tree $T$ and a set $\mathcal{G}''_{j,k}$ of dumbbells having the length grouping, empty region, and nesting properties. Every head of a dumbbell in $\mathcal{G}''_{j,k}$ is the bounding rectangle of a node in $T$.

**Output:** A dumbbell tree $T_{\mathcal{G}''_{j,k}}$ of $\mathcal{G}''_{j,k}$.

1: Let $\mathcal{G}'_{1,j,k}, \ldots, \mathcal{G}'_{t,j,k}$ be the constituent subgroups of $\mathcal{G}''_{j,k}$. Then assign a label $\lambda(D(A,B)) = i$ to every dumbbell $D(A,B) \in \mathcal{G}'_{i,j,k}$.
2: Scan the list of dumbbells in $\mathcal{G}''_{j,k}$, and add every dumbbell $D(A,B) \in \mathcal{G}''_{j,k}$ as a node to $T_{\mathcal{G}''_{j,k}}$.
3: For every node $A \in T$, let $\mathcal{D}(A)$ be the set of dumbbells $D(A,B_1), \ldots, D(A,B_x)$ which have $R(A)$ as one of their heads, sorted by their labels $\lambda(D(A,B_1)), \ldots, \lambda(D(A,B_x))$.
4: Process tree $T$ from the root toward the leaves to compute for every node $A \in T$, the lowest proper ancestor $A'$ so that $\mathcal{D}(A') \neq \emptyset$, and set $\kappa(A) = (\lambda(D(A',B'_1)), R_{B'_1}(A'))$, where $D(A',B'_1)$ is the first entry in $\mathcal{D}(A')$.
5: **for** every internal node $A \in T$ **do**
6: Let $\mathcal{D}(A) = \{D(A,B_1), \ldots, D(A,B_x)\}$.
7: Create heads $R_{B_1}(A), \ldots, R_{B_x}(A)$ with parents $D(A,B_1), \ldots, D(A,B_x)$, respectively.
8: **for** $i = 1, \ldots, x-1$ **do**
9: Append a triple $(D(A,B_i), \lambda(D(A,B_{i+1})), R_{B_{i+1}}(A))$ to a list $P$ of potential parents for all dumbbells.
10: **end for**
11: Append a triple $(D(A,B_x), \lambda(D(A',B'_1)), R_{B'_1}(A'))$ to the list $P$ of potential parents, where $\kappa(A) = (\lambda(D(A',B'_1)), R_{B'_1}(A'))$.
12: **end for**
13: Sort list $P$ lexicographically, thereby storing triples $(D(A,B), \lambda_1(A,B), R_1(A,B))$ and $(D(A,B), \lambda_2(A,B), R_2(A,B))$ consecutively, for every dumbbell $D(A,B)$.
14: Scan list $P$ and set $p(D(A,B)) = R_i(A,B)$, where $\lambda_i(A,B) = \min(\lambda_1(A,B), \lambda_2(A,B))$.
15: **for** every leaf $q$ of $T$ **do**
16: Add a node $q$ to $T_{\mathcal{G}''_{j,k}}$.
17: **if** $\mathcal{D}(q) \neq \emptyset$ **then**
18: $p(q) \leftarrow R_{B_1}(q)$, where $D(q,B_1)$ is the first entry in $\mathcal{D}(q)$.
19: **else**
20: $p(q) \leftarrow R_{B'_1}(A')$, where $\kappa(q) = (\lambda(D(A',B'_1)), R_{B'_1}(A'))$.
21: **end if**
22: **end for**

**Algorithm 13.3**
Computing a dumbbell tree of a group of dumbbells having the length grouping, empty region, and nesting properties.

**Lemma 13.13** *Algorithm 13.3 computes a dumbbell tree $T_{\mathcal{G}''_{j,k}}$ of $\mathcal{G}''_{j,k}$, provided that $\delta < \frac{1}{2\sqrt{d}}$.*

*Proof.* It is easily verified that there is one node per dumbbell, dumbbell head, or leaf of $T$ in the constructed tree $T_{\mathcal{G}''_{j,k}}$. Every dumbbell head is the child of its corresponding dumbbell. We have to show that the parent of every leaf and dumbbell node is computed correctly.

For a leaf $p$, the lowest ancestor $A$ is found so that $\mathcal{D}(A) \neq \emptyset$. Ancestor $A$ exists because $\mathcal{D}(S) \neq \emptyset$, for the root $S$ of $T$. The dumbbells in $\mathcal{D}(A)$ are sorted by increasing length groups. By the empty region property, there are no two dumbbells $D(A, B_1)$ and $D(A, B_2)$ with head $R(A)$ in the same length group of $\mathcal{G}''_{j,k}$. Hence, the first dumbbell $D(A, B) \in \mathcal{D}(A)$ is the shortest dumbbell with head $R(A)$. For any ancestor $A'$ of $A$ and any dumbbell $D(A', B') \in \mathcal{D}(A')$, $\lambda(D(A, B)) < \lambda(D(A', B'))$, by the empty region property and Lemma 13.12. Thus, $D(A, B)$ is the shortest dumbbell containing point $p$, and head $R_B(A)$, as computed by the algorithm, is the correct parent of point $p$.

For a dumbbell $D(A, B)$, the argument is similar. In particular, we show that the potential parents $p'(R_B(A))$ and $p'(R_A(B))$ of dumbbell $D(A, B)$ are chosen correctly from among the bounding rectangles of the ancestors of nodes $A$ and $B$ in $T$. Since $p(D(A, B))$ is chosen from $p'(R_B(A))$ and $p'(R_A(B))$ so that the dumbbell having $p(D(A, B))$ as a head is in the lower length group, this implies that $p(D(A, B))$, as computed by the algorithm, is the correct parent of dumbbell node $D(A, B)$.

So consider head $R_B(A)$. The proof for head $R_A(B)$ is similar. Let $p'(R_B(A)) = R_{B'}(A')$. If $A = A'$, then $D(A', B')$ is the immediate successor of $D(A, B)$ in $\mathcal{D}(A)$. By the empty region property, there are no two dumbbells with the same head in the same length group of $\mathcal{G}''_{j,k}$, so that $\lambda(D(A'', B'')) < \lambda(D(A, B))$, for every dumbbell $D(A'', B'')$ preceding $D(A, B)$ in $\mathcal{D}(A)$, and $\lambda(D(A'', B'')) > \lambda(D(A', B'))$, for every dumbbell $D(A'', B'')$ succeeding $D(A', B')$ in $\mathcal{D}(A)$. For every ancestor $A''$ of $A$ and every dumbbell $D(A'', B'') \in \mathcal{D}(A'')$, $\lambda(D(A'', B'')) > \lambda(D(A', B'))$, by the empty region property and Lemma 13.12. Hence, $\lambda(D(A', B')) = \min\{\lambda(D(A'', B'')) :$

$R_B(A) \subseteq R_{B''}(A'')$ and $\lambda(D(A'', B'')) > \lambda(D(A, B))\}$. That is, $R_{B'}(A')$ is the proper potential parent of $D(A, B)$ w.r.t. $R_B(A)$ in $T_{\mathcal{G}''_{j,k}}$.

If $A \neq A'$, $D(A, B)$ is the last dumbbell in $\mathcal{D}(A)$, $A'$ is the lowest ancestor of $A$ in $T$ so that $\mathcal{D}(A') \neq \emptyset$, and $D(A', B')$ is the first dumbbell in $\mathcal{D}(A')$. By the empty region property, $\lambda(D(A, B'')) < \lambda(D(A, B))$, for every dumbbell $D(A, B'') \neq D(A, B)$ with head $R(A)$, and $\lambda(D(A', B')) < \lambda(D(A', B''))$, for every dumbbell $D(A', B'') \neq D(A', B')$ with head $R(A')$. By the empty region property and Lemma 13.12, $\lambda(D(A', B')) < \lambda(D(A'', B''))$, for every ancestor $A''$ of node $A'$ and every dumbbell $D(A'', B'') \in \mathcal{D}(A'')$. Since there is no node $A''$ with $\mathcal{D}(A'') \neq \emptyset$ which is a proper ancestor of $A$ and a proper descendant of $A'$, this shows that $\lambda(D(A', B')) = \min\{\lambda(D(A'', B'')) : R_B(A) \subseteq R_{B''}(A'')$ and $\lambda(D(A'', B'')) > \lambda(D(A, B))\}$, so that $R_{B'}(A')$ is the proper potential parent of $D(A, B)$ w.r.t. $R_B(A)$ in this case as well. □

**Lemma 13.14** *Algorithm 13.3 takes $\mathcal{O}(\text{sort}(N))$ I/Os and uses linear space.*

*Proof.* As a result of the sorting step at the end of Algorithm 13.2, the dumbbells in $\mathcal{G}''_{j,k}$ are already sorted according to their membership in groups $\mathcal{G}'_{1,j,k}, \ldots, \mathcal{G}'_{t,j,k}$. Moreover, every dumbbell $D(A, B)$ bears a label $\gamma(D(A, B)) = (j, k, l)$, where $l$ identifies the length group containing dumbbell $D(A, B)$. Hence, labels $\lambda(D(A, B))$, $D(A, B) \in \mathcal{G}''_{j,k}$ can be produced in a single scan of the list of dumbbells in $\mathcal{G}''_{j,k}$. We construct sets $\mathcal{D}(A)$, $A \in T$, as follows: We scan the list of dumbbells $D(A, B) \in \mathcal{G}''_{j,k}$ and generate two triples $(A, \lambda(D(A, B)), D(A, B))$ and $(B, \lambda(D(A, B)), D(A, B))$, for every dumbbell $D(A, B) \in \mathcal{G}''_{j,k}$. Then we sort this list lexicographically, thereby storing all dumbbells having head $R(A)$ consecutively, sorted according to their length group labels $\lambda(D(A, B))$. Lines 4–12 and Lines 15–22 can be carried out in a single pass of time-forward processing from the root toward the leaves in $T$. Line 13 sorts the list $P$ of triples representing potential parents of the dumbbells in $\mathcal{G}''_{j,k}$. Line 14 scans this list. □

The following result now follows from Lemmas 13.3, 13.4, 13.13, and 13.14.

**Theorem 13.3** *Given a set $S$ of $N$ points in $\mathbb{R}^d$ and a WSPD $\mathcal{D} = (T, \mathcal{R})$ of $S$, it takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os and linear space to partition the set of dumbbells induced by realization $\mathcal{R}$ into $\mathcal{O}(1)$ groups, each having the length grouping, empty region, and nesting properties. Each of these groups can be represented by a dumbbell tree. The construction of all dumbbell trees takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os.*

# Chapter 14

# A Planar Steiner Spanner

The dumbbell spanner discussed in the previous chapter allows spanner paths to be reported efficiently, and the WSPD spanner provides the underlying structure for the dumbbell spanner. Unfortunately it seems that neither of the two spanners can be constructed in constrained situations where only a subset of the edges of the complete graph are allowed to be in the spanner. An important special case is the construction of a spanner for a set of polygonal obstacles, where such a spanner can be used to compute $(1+\varepsilon)$-shortest paths among these obstacles. In [125], we show that the $\theta$-graph can be constructed I/O-efficiently for sets of polygonal obstacles in the plane; but we were not able to design a data structure that allows spanner path queries to be answered quickly.

In this chapter, we show that the planar Steiner spanner of [16] can be constructed I/O-efficiently for a set of polygonal obstacles in the plane. The graph has the desirable property to be planar, so that we can use Theorem 10.2 to find shortest paths in the spanner. We can also combine the results of Chapters 8 and 10 with existing results [2, 100, 177] to preprocess the graph in $\mathcal{O}(\mathrm{sort}(N))$ I/Os so that shortest path queries can be answered and paths in the graph can be traversed, both in an I/O-efficient manner.

Since it is easy to show that in general there is no planar spanner with spanning ratio $t < \sqrt{2}$ for a point set in the plane, the spanner construction of [16] adds *Steiner*

*points* to the point set. These are points which are not in the given point set and whose locations are chosen so that the resulting point set has a planar $t$-spanner.

The solution presented here follows the framework of the algorithm proposed in [16]. However, the details differ. In particular, the data structure used to maintain the sweep-line status of the algorithm in Section 14.2 differs considerably from the one used in [16] and also leads to a simplified internal memory algorithm. Since no journal version of [16] has appeared and according to one of the authors is unlikely to appear because a number of co-authors have left academia, we present detailed proofs for lemmas that are stated, but not proved in [16].

In Section 14.1, we show how to construct a planar $L_1$-Steiner spanner for a point set. In Section 14.2, we present an algorithm for constructing a planar $L_1$-Steiner spanner of a set of polygonal obstacles. Both constructions follow the same framework: First we construct an appropriate planar subdivision. Then we add edges and Steiner points inside every region of the subdivision to obtain a graph with $L_1$-spanning ratio $1 + \varepsilon$. While the subdivision used to construct a planar $L_1$-Steiner spanner for a point set can be obtained quite easily, considerable effort is required to compute the subdivision for a set of obstacles efficiently.

In Section 14.3, we use the fact that planar $L_1$-Steiner spanners can be computed I/O-efficiently in order to derive a planar $L_2$-Steiner spanner as the superimposition of a constant number of $L_1$-Steiner spanners.

## 14.1 A Planar $L_1$-Steiner Spanner for Point Sets

As explained above, to construct a planar $L_1$-Steiner spanner for a point set $S$, we first compute an appropriate planar subdivision defined by $S$ and then introduce additional vertices and edges inside every region of this subdivision.

In Section 14.1.1, we define this subdivision and show that it is easily derived from a fair split tree of point set $S$. In Section 14.1.2, we use the subdivision to construct the spanner.

### 14.1.1 A Planar Subdivision

Let $S$ be the given point set. The subdivision $D'$ we use in the spanner construction is a subdivision of a minimal square $C$ containing all points of $S$ into $\mathcal{O}(N)$ regions of two types: *box cells* and *donut cells*. A box cell is a box and contains exactly one point in $S$. A donut cell is the set-theoretic difference of two boxes $R$ and $R'$ which does not contain any points in $S$. Box $R'$ is contained in box $R$. The distance of each side $e'$ of $R'$ from the corresponding side of $R$ is either zero or at least $\|e'\|_1/3$.

Such a subdivision $D' = D'(T)$ can be obtained quite naturally from a fair split tree $T$ of $S$: For every leaf $p$ of $T$, we add the split rectangle $\tilde{R}(p)$ as a cell to $D'(T)$. For every internal node $A$ of $T$ with $\hat{R}(A) \neq \tilde{R}(A)$, we add the region $\tilde{R}(A) \setminus \hat{R}(A)$ as a cell to $D'(T)$.

**Lemma 14.1** *The collection $D'(T)$ of regions defines a subdivision of $C$ whose regions are either box or donut cells.*

*Proof.* The fact that $D'(T)$ is a subdivision of $C$ is easily shown by induction. In particular, the rectangle $\tilde{R}(p)$ is a trivial subdivision of rectangle $\tilde{R}(p)$, for every leaf $p$ of $T$. Now let $A$ be an internal node with children $A_1$ and $A_2$. By the induction hypothesis, the cells introduced by descendants of $A_1$ and $A_2$ define subdivisions of rectangles $\tilde{R}(A_1)$ and $\tilde{R}(A_2)$. Hence, their union is a subdivision of $\hat{R}(A)$ because $\hat{R}(A) = \tilde{R}(A_1) \cup \tilde{R}(A_2)$ and $\tilde{R}(A_1) \cap \tilde{R}(A_2) = \emptyset$. If $A$ is the root of $T$, we thus obtain the desired subdivision of square $C = \hat{R}(A)$. Otherwise, if $\tilde{R}(A) = \hat{R}(A)$, node $A$ does not contribute any cell to $D'(T)$. If $\tilde{R}(A) \neq \hat{R}(A)$, node $A$ contributes cell $\tilde{R}(A) \setminus \hat{R}(A)$ to $D'(T)$. In both cases, the cells introduced by descendants of $A$ form a subdivision of rectangle $\tilde{R}(A)$, thereby completing the inductive step.

Now observe that all rectangles $\tilde{R}(A)$ and $\hat{R}(A)$ are boxes. Hence, all cells $\tilde{R}(p)$, where $p$ is a leaf of $T$, are box cells, as they contain exactly one point, namely point $p$. Also, every cell $\tilde{R}(A) \setminus \hat{R}(A)$ is the set-theoretic difference of two boxes. Each cell $\tilde{R}(A) \setminus \hat{R}(A)$ is empty because $\hat{R}(A)$ is constructed from $\tilde{R}(A)$ by splitting off empty

rectangles. It remains to show that every side $e'$ of $\hat{R}(A)$ is either contained in the corresponding side $e$ of $\tilde{R}(A)$ or has distance at least $\|e'\|_1/3$ from $e$.

Let $\tilde{R}(A) = R_0, R_1, \ldots, R_k = \hat{R}(A)$ be the sequence of rectangles produced by the procedure constructing $\hat{R}(A)$ from $\tilde{R}(A)$. For a side $e'$ of $\hat{R}(A)$ that is not contained in the corresponding side $e$ of $\tilde{R}(A)$, let $R_i$ be the first rectangle in the above sequence so that the side $e_i$ of $R_i$ corresponding to $e$ is not contained in $e$. Then $R_i$ is one of the rectangles produced by splitting rectangle $R_{i-1}$ fairly, perpendicular to its longest side. Let $e_{i-1}$ be the side of rectangle $R_{i-1}$ corresponding to side $e_i$, and let $i_{\max} = i_{\max}(R_{i-1})$. Then the distance between $e_i$ and $e_{i-1}$ is at least $\frac{1}{3}\ell_{\max}(R_{i-1})$. Moreover, $\ell_{\max}(\hat{R}(A)) \leq \ell_{\max}(R_{i-1})$ and $\hat{R}(A) \subseteq R_i \subseteq R_{i-1} \subseteq \tilde{R}(A)$, so that the distance between $e$ and $e'$ is at least $\frac{1}{3}\ell_{\max}(\hat{R}(A)) \geq \|e'\|_1/3$. □

The construction of subdivision $D'(T)$ requires computing a fair split tree $T$ of $S$ and scanning the vertex set of $T$ to extract the cells of $D'(T)$. By Lemma 12.10, a fair split tree of size $\mathcal{O}(N)$ for $S$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. The extraction of the cells takes $\mathcal{O}(\text{scan}(N))$ I/Os. Every node of $T$ adds at most one cell to $D'(T)$, so that we obtain the following lemma.

**Lemma 14.2** *Subdivision $D'(T)$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space. Its size is $\mathcal{O}(N)$.*

### 14.1.2 The Spanner

Given a subdivision $D'$ as defined in Section 14.1.1, a planar $L_1$-spanner for point set $S$ can be constructed as follows: Let INTERVAL$(e, r)$ be a procedure which partitions segment $e$ into a minimum number of subsegments of length at most $r$ by adding equally spaced Steiner points on edge $e$. We perform INTERVAL$(e, \gamma\|e\|_1)$, for every boundary edge $e$ of a cell in the subdivision, where $\gamma$ is an appropriately chosen constant to be defined later. For every cell $R$ and every boundary edge $e$ of $R$, we shoot rays orthogonal to $e$ from the endpoints of $e$ and the Steiner points on $e$ toward the interior of $R$ until they hit another boundary edge of $R$. For every box cell $R$

containing a point $p \in S$, we also shoot rays from $p$ in all four axis-parallel directions until they hit the boundary of $R$. We add these rays as edges to $D'$. To preserve the planarity of the resulting graph, we introduce all intersection points between perpendicular rays as Steiner points. We call the resulting graph $D''$. The following two lemmas show that $D''$ is the desired spanner.

**Lemma 14.3 (Arikati et al. [16])** *Graph $D''$ has size $\mathcal{O}(N/\gamma^2)$.*

*Proof.* Subdivision $D'$ has size $\mathcal{O}(N)$, and each cell of $D'$ is bounded by at most eight edges, four horizontal and four vertical ones. Each edge $e$ is partitioned into at most $\lceil 1/\gamma \rceil$ smaller edges. That is, at most $4/\gamma + 8$ vertical rays and $4/\gamma + 8$ horizontal rays are shot into the cell. This creates at most $(4/\gamma + 8)^2 = \mathcal{O}(1/\gamma^2)$ intersection points per cell which are added as Steiner points to $D''$. As graph $D''$ is planar, the number of edges and faces is linear in the number of vertices, and the lemma follows. □

**Lemma 14.4 (Arikati et al. [16])** *The $L_1$-spanning ratio of graph $D''$ is at most $1 + 3\gamma$.*

*Proof.* Consider two points $a, b \in S$. Then segment $(a, b)$ is the shortest path from $a$ to $b$ in any metric. Replace $(a, b)$ by a path $(a, c, b)$ by moving first vertically from $a$ until a point $c$ with the same $y$-coordinate as $b$ is reached and then horizontally to $b$. Path $(a, c, b)$ has the same length as edge $(a, b)$ in the $L_1$-metric. Hence, it is sufficient to construct a path in $D''$ which is of length at most $(1 + 3\gamma)\|(a, c, b)\|_1$, where $\|(a, c, b)\|_1 = \|(a, c)\|_1 + \|(c, b)\|_1$ is the length of path $(a, c, b)$ in the $L_1$-metric. Let $p_1, \ldots, p_k$ be the intersection points between path $(a, c, b)$ and edges of $D'$ that are not contained in the same lines as segments $(a, c)$ and $(c, b)$. Then $\|(a, c, b)\|_1 = \|(a, p_1)\|_1 + \sum_{i=1}^{k-1} \|(p_i, p_{i+1})\|_1 + \|(p_k, b)\|_1$.

First observe that segments $(a, p_1)$ and $(p_k, b)$ coincide with edges of $D''$ incident to points $a$ and $b$. To show the existence of $(1+3\gamma)$-approximations of edges $(p_i, p_{i+1})$, $1 \leq i \leq k-1$, we prove the following claim: Let $p$ and $q$ be two points on the boundary of a cell $R$ of $D'$ so that there is an $L_1$-path from $p$ to $q$ which intersects the boundary

of $R$ only in $p$ and $q$ and has at most one bend. Then there exists a path from $p$ to $q$ in $D''$ whose length is at most $(1+3\gamma)\|(p,q)\|_1$. This implies that there exists a path from $a$ to $b$ in $D'$ whose length is at most $\|(a,p_1)\|_1 + (1+3\gamma)\sum_{i=1}^{k-1}\|(p_i,p_{i+1})\|_1 + \|(p_k,b)\|_1 \leq (1+3\gamma)\|(a,c,b)\|_1$ because every subpath of path $(a,b,c)$ between two consecutive points $p_i$ and $p_{i+1}$ has at most one bend. There are three possible cases (see Figure 14.1):

(a) $R$ is a box cell and points $p$ and $q$ are on two perpendicular sides of $R$, or $R$ is a donut cell and points $p$ and $q$ are on two perpendicular sides of the outer box of $R$,

(b) $R$ is a donut cell, $p$ is on the outer box of $R$, and $q$ is on the perpendicular side of the inner box of $R$ which is closer to $p$, or

(c) Points $p$ and $q$ can be connected by a horizontal or vertical line segment which stays completely inside $R$.

In Cases (a) and (b), the paths from $p$ to $q$ shown in Figure 14.1 have length $\|(p,q)\|_1$ and consist of edges in $D''$. In Case (c), let $e$ be the edge of $D'$ containing point $p$ and $e'$ be the edge containing point $q$, where w.l.o.g. $\|e'\|_1 \leq \|e\|_1$. Then a spanner path from $p$ to $q$ is obtained by walking along edge $e$ to the closest vertex $v$ of $D''$ on edge $e$, continuing perpendicular to edge $e$ toward the vertex $w$ on edge $e'$ which is closest to $q$ and then walking from $w$ to $q$ along edge $e'$. Since the vertices on edge $e'$ have distance at most $\gamma\|e'\|_1$, the distance between point $q$ and vertex $w$ is at most $\gamma\|e'\|_1/2$. The same is true for point $p$ and vertex $v$ because $\mathrm{dist}_1(p,v) = \mathrm{dist}_1(q,w)$. Hence, the constructed path has length at most $\|(v,w)\|_1 + \gamma\|e'\|_1 = \|(p,q)\|_1 + \gamma\|e'\|_1$. However, the distance between edges $e'$ and $e$ is at least $\|e'\|_1/3$, so that $\|e'\|_1 \leq 3\|(p,q)\|_1$. This implies that path $(p,v,w,q)$ has length at most $(1+3\gamma)\|(p,q)\|_1$. □

**Lemma 14.5** *Graph $D''$ can be constructed in $\mathcal{O}(\mathrm{sort}(N/\gamma^2))$ I/Os using $\mathcal{O}(N/(\gamma^2 B))$ blocks of external memory.*

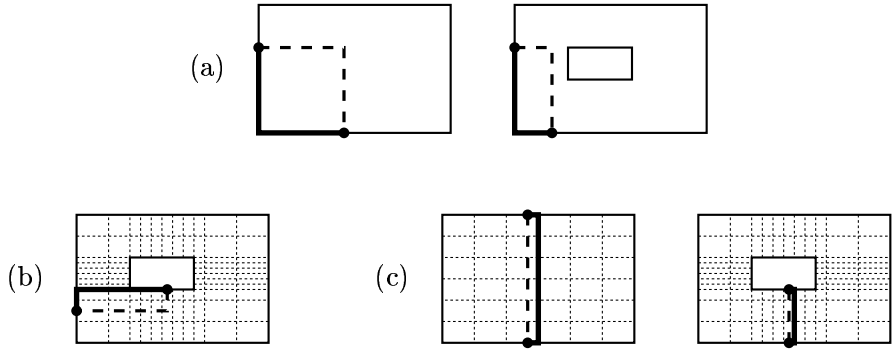



**Figure 14.1**
Spanner paths between points on the boundaries of cells of subdivision $D'$. The spanner paths are shown as fat solid lines. $L_1$-paths between the endpoints of these spanner paths are shown as fat dashed lines. Thin dotted lines are the rays used to partition the cells.

*Proof.* By Lemma 14.2, subdivision $D'$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space. Given the cells of $D'$, we scan the set of cells to partition each of them as described above and add the segments in the resulting partition to the edge set of $D''$ and their endpoints to the vertex set. Since each cell has a constant size description, we can load it into internal memory and generate the set of vertices and edges in this cell in a linear number of I/Os. By Lemma 14.3, the total number of vertices and edges is $\mathcal{O}(N/\gamma^2)$, so that generating them takes $\mathcal{O}(\text{scan}(N/\gamma^2))$ I/Os. Since each cell is partitioned separately, the resulting vertex set may contain duplicates, and the edge set may contain edges that overlap (see Figure 14.2a). The duplicates in the vertex set can be removed in $\mathcal{O}(\text{sort}(N/\gamma^2))$ I/Os using procedure DuplicateRemoval.

To construct the final edge set of $D''$, we have to replace overlapping edges as follows: Let $v_1 = u_1, \ldots, u_k = v_2$ be the sequence of endpoints of the first chain of edges, and let $v_1 = w_1, \ldots, w_l = v_2$ be the endpoints of edges in the second chain. Our goal is to construct a chain of edges with endpoints $v_1 = x_1, \ldots, x_q = v_2$ so that for all $1 \leq i \leq q$, there is a vertex $u_j$ or $w_{j'}$ such that $x_i = u_j$ or $x_i = w_{j'}$, for all $1 \leq j \leq k$, there is a vertex $x_i$ such that $u_j = x_i$, and for all $1 \leq j' \leq l$, there is a vertex $x_i$ such that $w_{j'} = x_i$.

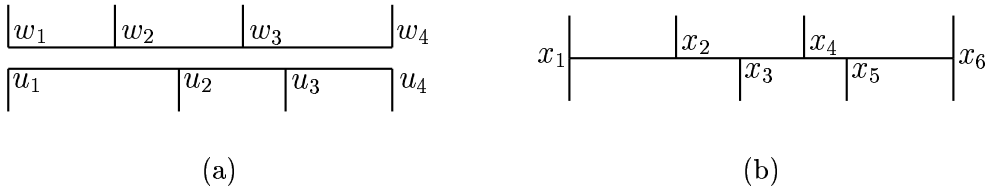



**Figure 14.2**
Superimposing overlapping edges.

We obtain this superimposition of overlapping edges as follows: We consider only horizontal edges. Vertical edges can be handled analogously. We sort the horizontal edges by their $y$-coordinates and the $x$-coordinates of their left endpoints. Then a single scan of this sorted edge list is sufficient to generate edges $\{x_1, x_2\}, \ldots, \{x_{q-1}, x_q\}$ as defined above. Hence, the final edge set of $D''$ can be obtained in $\mathcal{O}(\text{sort}(N/\gamma^2))$ I/Os.

□

If we choose $\gamma = \varepsilon/3$, Lemmas 14.3, 14.4, and 14.5 lead to the following result.

**Theorem 14.1** *Given a point set $S$ in the plane and a constant $\varepsilon > 0$, a planar Steiner spanner of size $\mathcal{O}(N/\varepsilon^2)$ and with $L_1$-spanning ratio $1 + \varepsilon$ can be computed in $\mathcal{O}(\text{sort}(N/\varepsilon^2))$ I/Os using $\mathcal{O}(N/(\varepsilon^2 B))$ blocks of external memory.*

## 14.2 A Planar $L_1$-Steiner Spanner for Sets of Polygonal Obstacles

In this section, we adapt the construction of the previous section to construct a planar $L_1$-spanner for a set $P$ of polygonal obstacles in the plane. That is, we construct a planar subdivision similar to the one described in Section 14.1.1 and then derive a spanner from the subdivision in a manner similar to the construction in Section 14.1.2. But now the subdivision is constrained by the set of obstacle edges, which makes it more difficult to compute. In order to compute the subdivision, we again follow the framework of an algorithm of [16], which employs a plane-sweep to carry out its

task. However, our representation of the sweep-line status uses only a single buffer tree instead of two balanced search trees. This simplification is the key to obtaining an I/O-efficient algorithm because the approach of [16] requires immediate query responses on the two trees; but no I/O-optimal batched search structure exists which answers queries immediately.

In Section 14.2.1, we define the subdivision used to obtain the spanner. In Section 14.2.2, we show that the subdivision defined in Section 14.2.1 can be used to derive a planar $L_1$-Steiner spanner for obstacle set $P$ in a manner similar to the construction in Section 14.1.2. In Sections 14.2.3 and 14.2.4, we present an I/O-efficient algorithm for computing the subdivision defined in Section 14.2.1.

### 14.2.1 A Modified Planar Subdivision

First we compute a modified planar subdivision so that we can obtain an $L_1$-spanner of $P$ by partitioning the cells of this subdivision as in Section 14.1.2 and removing all edges that are inside obstacles. It is not hard to see that the following subdivision $D_1$ has this property: Let $D$ be the planar subdivision induced by the obstacles in $P$, and let $D'$ be the subdivision obtained from the set of obstacle vertices using the construction in Section 14.1.1. Then $D_1$ is the superimposition of $D$ and $D'$. Unfortunately, $D_1$ may have size $\Omega(N^2)$, thereby leading to a spanner of superlinear size. In [16], it is shown that another subdivision $D_2$ with the desired properties can be derived from $D_1$ by removing edges between cells. This subdivision has linear size and can be constructed without constructing $D_1$ explicitly. In this section, we recall the definition of this subdivision.

**Subdivision $D_1$.** To define subdivision $D_2$, we first define the superimposition $D_1$ of $D$ and $D'$ formally and study its structure. Let $S$ be the vertex set of the obstacles in $P$, and let $D' = (S', E')$ be the subdivision $D'$ defined for point set $S$ in Section 14.1.1, viewed as a graph. Let $D = (S, E)$ be the graph defined by the set of obstacles in $P$. Then let graph $D_1$ be the superimposition of graphs $D'$ and $D$. That

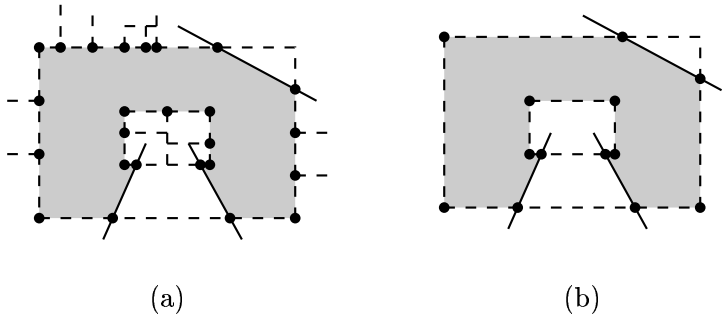



**Figure 14.3**
A face with many vertices on its boundary (a) and its corresponding region (b).

is, the edges of $D'$ and $D$ are split at their intersection points, and these intersection points are introduced as vertices of $D_1$.

In order to derive subdivision $D_2$ from $D_1$, we need to distinguish between a *region* of subdivision $D_1$ and a *face* of graph $D_1$. Every face $f$ of graph $D_1$ is contained in a region $R$ of subdivision $D'$. We define the boundary of region $R(f)$ corresponding to face $f$ as the sequence of edges defined by boundary vertices of region $R$, obstacle vertices on the boundary of face $f$, and intersection points between the boundary of region $R$ and obstacle edges. As a result, the complexity of every region in subdivision $D_1$ is bounded, even though the faces of graph $D_1$ may have a large number of vertices on their boundaries (see Figure 14.3).

The regions of subdivision $D_1$ can now be partitioned into two classes: A *red* region is a quadrilateral $R(f)$ so that no vertex of face $f$ is in $S \cup S'$ (Figure 14.4a). That is, all the vertices of face $f$ are intersection points between edges in $E$ and $E'$. In particular, the edges of such a region alternate between $E$ and $E'$ in clockwise order around the region. All remaining regions are *blue* (Figure 14.4b).

A blue region $R(f)$ can be of two types, depending on whether face $f$ has a vertex in $S \cup S'$. If face $f$ has no vertex in $S \cup S'$ on its boundary, region $R(f)$ is bounded by six or eight edges. This follows from the following three observations: If it had only four edges on its boundary, it would be red. The number of boundary edges must be even because edges from $E$ and $E'$ alternate along the boundary of $R(f)$. There

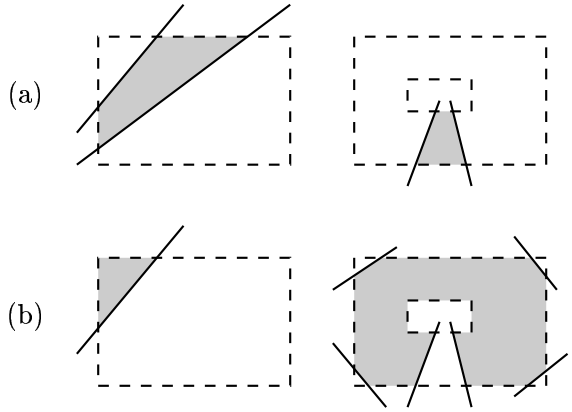



**Figure 14.4**
(a) Two red regions. (b) The simplest and the most complicated blue region.

cannot be more than four edges from $E'$ on the boundary of $R(f)$ if $R(f)$ does not contain a vertex from $S \cup S'$. If face $f$ has a vertex in $S \cup S'$, the shape of $R(f)$ can vary. However, region $R(f)$ can be bounded by at most 16 edges: 6 obstacle edges and 10 edges from $E'$. Hence, blue and red regions of $D_1$ have constant complexity. The construction of subdivision $D_2$ maintains this property for all its regions. This is important for the construction of the Steiner spanner from $D_2$.

**Ladders, rungs, and the red graph.** The construction of $D_2$ merges adjacent red regions in $D_1$, where adjacency is defined by means of a *red graph* of subdivision $D_1$. This graph is a subgraph of the dual of $D_1$. It contains a vertex for every red region of $D_1$ and an edge between two vertices if the two corresponding regions share an edge that is part of an edge in $E'$. Since every red region has two edges from $E'$ on its boundary, every vertex in the red graph has degree at most two. Hence, every connected component of the red graph is a path (Figure 14.5).

Consider the set of red regions corresponding to a connected component of the red graph. These regions are bounded by the same two obstacle edges and a set of edges in $E'$. We call such a set of red regions a *ladder*. The two obstacle edges on their boundaries are the *sides* of the ladder; the edges from $E'$ are its *rungs*. More generally, the rungs of graph $D_1$ are all edges in $D_1$ that are contained in edges of $D'$

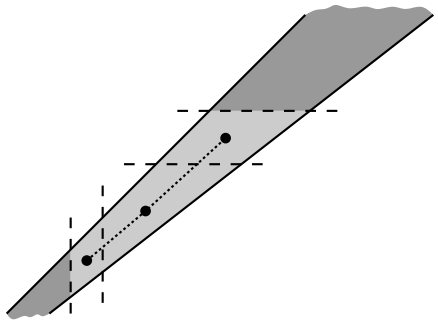


**Figure 14.5**
A non-trivial ladder and its corresponding connected component of the red graph.

and both of whose endpoints are intersection points of edges in $E$ and $E'$. We call the topmost rung of a ladder its *top rung*. *Left*, *right*, and *bottom rungs* are defined in a similar manner. All of these four types of rungs are called *extremal rungs*. To simplify the description of the algorithm for constructing $D_2$, we also consider a single rung between two blue regions to be a ladder. We call such a ladder *trivial*, while a ladder formed as the union of red regions is *non-trivial*.

**Subdivision $D_2$.** Subdivision $D_2$ is now obtained by removing the non-extremal rungs from all ladders of $D_1$. That is, the red regions of each non-trivial ladder are merged into a single red region. The following lemma shows that subdivision $D_2$ has linear size.

**Lemma 14.6 (Arikati et al. [16])** *Subdivision $D_2$ has size $\mathcal{O}(N)$.*

*Proof.* The number of blue regions each having at least one vertex in $S \cup S'$ on its boundary is $\mathcal{O}(N)$ because $|S| = N$, $|S'| = \mathcal{O}(N)$, every vertex in $S$ is on the boundary of two regions, and every vertex in $S'$ is on the boundary of at most four regions. The number of blue regions whose vertices are intersection points of $D$ and $D'$ and the number of red regions is linear in the number of rungs of $D_2$. We show that there are $\mathcal{O}(N)$ top rungs. Using the same arguments, it can be shown that the number of left, right, and bottom rungs is linear. Hence, the total number of extremal rungs is $\mathcal{O}(N)$, and subdivision $D_2$ has $\mathcal{O}(N)$ regions.

In order to bound the number of top rungs in subdivision $D_2$, we consider the trapezoidation $\Delta$ of the obstacle set $P$ obtained by shooting horizontal rays from all obstacle vertices. This trapezoidation contains $\mathcal{O}(N)$ trapezoids. For every trapezoid $f$, let $S'_f$ be the set of vertices of subdivision $D'$ that lie in trapezoid $f$. We show that the number of top rungs in trapezoid $f$ is at most $1 + |S'_f|$, so that the total number of top rungs is $\sum_{f \in \Delta}(1 + S'_f) = |\Delta| + |S'| = \mathcal{O}(N)$.

To prove the bound on the number of top rungs in a trapezoid $f$, let $e_1$ and $e_2$ be two top rungs in trapezoid $f$, and let $l$ and $r$ be the two obstacle edges bounding trapezoid $f$ on the left and right. The trapezoid $f'$ defined by edges $e_1$, $e_2$, $l$, and $r$ does not have any obstacle vertex on its boundary. Thus, if $f'$ does not contain a vertex in $S'_f$, it is the union of a set of red regions in $D_1$, and the bottom edge of $f'$ cannot be a top rung. This proves that any two consecutive top rungs in trapezoid $f$ have to be separated by at least one vertex in $S'_f$, and the bound on the number of top rungs in $f$ follows. □

Before showing in Sections 14.2.3 and 14.2.4 that subdivision $D_2$ can be constructed I/O-efficiently, we demonstrate that a planar $L_1$-Steiner spanner for obstacle set $P$ can be derived from $D_2$.

### 14.2.2 The Spanner

The procedure for deriving a spanner for obstacle set $P$ from subdivision $D_2$ is similar to the one presented in Section 14.1.2: We remove all regions inside obstacles from $D_2$. For each remaining region $R$, we consider all edges on its boundary that are contained in edges of $E'$. For each such edge $e$, we apply $\textsc{Interval}(e, \gamma\|e'\|_1)$, where $e'$ is the edge of subdivision $D'$ containing edge $e$, and shoot rays from the endpoints of $e$ and from the resulting Steiner vertices toward the interior of $R$ until they hit another boundary edge of $R$. Again, we add all intersection points between these rays as Steiner points to $D_2$. Let $D''$ be the resulting graph. The construction explicitly ensures that $D''$ is planar. The next two lemmas show that its size is small and its spanning ratio is at most $1 + 3\gamma$.

**Lemma 14.7 (Arikati et al. [16])** *Graph $D''$ has size $\mathcal{O}(N/\gamma^2)$.*

*Proof.* Since graph $D''$ is planar, the number of edges and faces of $D''$ is linear in the number of its vertices. Hence, it suffices to bound the number of vertices. By Lemma 14.6, subdivision $D_2$ has size $\mathcal{O}(N)$. Every region $R$ of $D_2$ is contained in a donut or box cell of subdivision $D'$, so that at most $4/\gamma + 8$ horizontal and vertical rays are shot into region $R$. As in the proof of Lemma 14.3, this implies that at most $(4/\gamma+8)^2 = \mathcal{O}(1/\gamma^2)$ Steiner points are introduced per region of $D_2$. Hence, the total number of vertices in $D''$ is $\mathcal{O}(N/\gamma^2)$. □

**Lemma 14.8 (Arikati et al. [16])** *Graph $D''$ has $L_1$-spanning ratio at most $1 + 3\gamma$.*

*Proof.* The proof is similar to that of Lemma 14.4. Let $a$ and $b$ be two obstacle vertices, and let $Q = (a = p_0, p_1, \ldots, p_k = b)$ be the shortest path from $a$ to $b$, where $p_1, \ldots, p_{k-1}$ are all bends and intersection points of $Q$ with edges in $E'$. W.l.o.g., we can assume that bends occur only at obstacle vertices and intersection points of $Q$ with edges of subdivision $D'$. In addition, we ensure that path $Q$ visits an obstacle vertex if without visiting it, it would come too close to that vertex. Formally, let $R$ be a box cell crossed by path $Q$, and let $p$ be the obstacle vertex contained in $R$. Let $q$ and $r$ be the two points on the boundary of $R$ crossed by $Q$. If $x(q) \leq x(p) \leq x(r)$ and $y(q) \leq y(p) \leq y(r)$, we ensure that path $Q$ contains point $p$ as one of its vertices. This can be done without changing the length of path $Q$ in the $L_1$-metric.

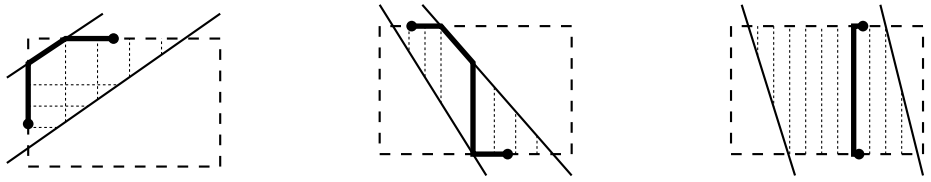
As in the proof of Lemma 14.4, it suffices to show that every edge $(p_{i-1}, p_i)$, $1 \leq i \leq k$, can be replaced with a path in $D''$ whose length is at most $(1+3\gamma)\|(p_{i-1}, p_i)\|_1$. We distinguish between the possible types of region $R$ containing edge $(p_{i-1}, p_i)$. Assume first that region $R$ is blue.

If w.l.o.g. $p_{i-1}$ is an obstacle vertex, then $p_i$ is an intersection point of $Q$ with an edge in $E'$ because no two obstacle vertices lie in the same region. No matter on which boundary edge of region $R$ point $p_i$ lies, it is easy to verify that there is a path of $L_1$-length $\|(p_{i-1}, p_i)\|_1$ from $p_{i-1}$ to $p_i$ in $D''$.

So assume that $p_{i-1}$ and $p_i$ are both intersection points. Then we construct a path $Q'$ as in the proof of Lemma 14.4, assuming that there are no obstacle edges. Path $Q'$ has length at most $(1+3\gamma)\|(p_{i-1},p_i)\|_1$, by Lemma 14.4. However, it may intersect obstacle edges and contain edges that are not in $D''$ because they are inside obstacles. We form pairs of consecutive intersection points between $Q'$ and obstacle edges on the boundary of region $R$ and replace the subpath of $Q'$ between each of these pairs $\{x,y\}$ of intersection points by the shortest path from $x$ to $y$ along the boundary of the obstacle containing $x$ and $y$. Let $Q''$ be the resulting path. Then we claim that $Q''$ uses only edges in $D''$ and $\|Q''\|_1 = \|Q'\|_1 \leq (1+3\gamma)\|(p_{i-1},p_i)\|_1$.

To prove the first claim, it suffices to prove that every pair $\{x,y\}$ of intersection points lies on the boundary of the same obstacle, so that the shortest path from $x$ to $y$ along the boundary of this obstacle exists. To see this, observe that when path $Q'$ leaves region $R$ through point $x$ on the boundary of an obstacle $o \in P$, it is confined to the intersection of obstacle $o$ with the cell $R'$ of subdivision $D'$ which contains $R$ because it never leaves cell $R'$. Hence, it can only re-enter region $R$ through an edge of the same obstacle, so that point $y$ is on the boundary of obstacle $o$.

The second claim follows if we can show that the shortest path in $D$ between two consecutive intersection points $x$ and $y$ is an edge. So assume for the sake of contradiction that there are two intersection points $x$ and $y$ so that the shortest path in $D$ from $x$ to $y$ is not an edge. Then region $R$ is contained in a box cell, which contains a vertex $p \in S$, and path $Q'$ intersects both edges $e_1$ and $e_2$ incident to vertex $p$. Now consider the three cases in the proof of Lemma 14.4. In Cases (a) and (b), path $Q'$ stays inside the axes-parallel rectangle $R'$ defined by points $p_{i-1}$ and $p_i$. Since path $Q$ contains a straight edge from $p_{i-1}$ to $p_i$, the definition of path $Q$ implies that point $p$ lies outside $R'$. Since edge $(p_{i-1},p_i)$ cannot intersect edges $e_1$ or $e_2$, this implies that edges $e_1$ and $e_2$ do not intersect rectangle $R'$, so that path $Q'$ does not intersect either of these two edges. This leads to the desired contradiction.



(a) (b) (c)

**Figure 14.6**
Spanner paths in different types of red regions.

In Case (c), assume first that points $p_{i-1}$ and $p_i$ are contained in the same slab defined by the rays shot perpendicular to the edges containing points $p_{i-1}$ and $p_i$. Then this slab does not contain point $p$ because this point itself defines one of these slab boundaries. Hence, path $Q'$ cannot intersect edges $e_1$ or $e_2$. If $p_{i-1}$ and $p_i$ lie in different slabs, path $Q'$ contains a slab boundary between the two slabs containing $p_{i-1}$ and $p_i$. This slab boundary is contained in rectangle $R'$, so that again the choice of path $Q$ guarantees that edges $e_1$ and $e_2$ do not intersect this slab boundary. Thus, we obtain a contradiction in Case (c) as well, which shows that the shortest path in $D$ between two consecutive intersection points $x$ and $y$ is an edge. This finishes the proof for edges $(p_{i-1}, p_i)$ that are contained in blue regions.

Now assume that edge $(p_{i-1}, p_i)$ is contained in a red region. In this case, points $p_{i-1}$ and $p_i$ lie on two opposite rungs of the region. If these two rungs are perpendicular to each other, the path shown in Figure 14.6a has length $\|(p_{i-1}, p_i)\|_1$. If the two rungs are parallel to each other, we distinguish two cases. Assume w.l.o.g. that the two rungs are horizontal. If w.l.o.g. point $p_i$ does not fall into the $x$-range of the rung containing $p_{i-1}$, the path shown in Figure 14.6b has length $\|(p_{i-1}, p_i)\|_1$. Otherwise, we construct a path from $p_{i-1}$ to $p_i$ as in the proof of Case (c) of Lemma 14.4 (Figure 14.6c). The fact that the constructed path has length at most $(1+3\gamma)\|(p_{i-1}, p_i)\|_1$ follows from the same argument as in that proof, observing that either the two edges of $R$ containing points $p_{i-1}$ and $p_i$ are opposite edges of the same region of subdivision $D'$, or they are even further apart. □

The previous two lemmas show that graph $D''$ is an $L_1$-Steiner spanner for obstacle set $P$. The following lemma shows that it can be constructed I/O-efficiently.

**Lemma 14.9** *Given subdivision $D_2$, a planar $L_1$-Steiner spanner of size $\mathcal{O}(N/\gamma^2)$ and with spanning ratio at most $1+3\gamma$ can be computed in $\mathcal{O}(\mathrm{sort}(N/\gamma^2))$ I/Os using $\mathcal{O}(N/(\gamma^2 B))$ blocks of external memory.*

*Proof.* Similar to the construction of spanner $D''$ for a point set $S$, the set of Steiner points and incident edges can be generated in internal memory, for every region of $D_2$, because each such region has constant complexity. Hence, a scan of the set of regions suffices to generate all Steiner points and edges in $\mathcal{O}(\mathrm{scan}(N/\gamma^2))$ I/Os.

An application of procedure DuplicateRemoval to the set of endpoints of the generated edges produces the final vertex set of graph $D''$. The generation of the edge set of $D''$ is complicated by the fact that edges are not only horizontal and vertical, as in the proof of Lemma 14.5. However, all generated edges are either part of an edge in $D'$ or obstacle edge, or they are new edges generated by partitioning the regions of $D_2$. The latter edges can be added to the edge set of $D''$ without modification. The former edges need to be superimposed as in the proof of Lemma 14.5. This superimposition can be performed using the same procedure as in the proof of Lemma 14.5, the only difference being that the edges are sorted by the identity of the obstacle edges or edges of $D'$ containing them. All edges that are contained in the same edge are sorted by the distances of their closer endpoints to one of the endpoints of $e$. Hence, graph $D''$ can be constructed in $\mathcal{O}(\mathrm{sort}(N/\gamma^2))$ I/Os using $\mathcal{O}(N/(\gamma^2 B))$ blocks of external memory. □

If we choose $\gamma = \varepsilon/3$, the following result follows from Lemmas 14.7, 14.8, 14.9, and 14.11. Proving Lemma 14.11 is the subject of Sections 14.2.3 and 14.2.4.

**Theorem 14.2** *Given a set $P$ of polygonal obstacles with a total of $N$ vertices, a planar $L_1$-Steiner spanner of size $\mathcal{O}(N/\varepsilon^2)$ and with spanning ratio at most $1+\varepsilon$ for $P$ can be computed in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB} + \mathrm{sort}(N/\varepsilon^2)\right)$ I/Os using $\mathcal{O}\left(\frac{N}{B}\left(\log_{\frac{M}{DB}}\frac{N}{DB} + \frac{1}{\varepsilon^2}\right)\right)$ blocks of external memory.*

### 14.2.3 Computing the Subdivision

Having shown that subdivision $D_2$ can be used to derive an $L_1$-Steiner spanner with spanning ratio $1+\varepsilon$ for obstacle set $P$, we have to provide an I/O-efficient algorithm for constructing subdivision $D_2$. As mentioned before, subdivision $D_1$ may have size $\Omega(N^2)$, so that constructing $D_1$ and removing all non-extremal rungs from the ladders of $D_1$ does not lead to an efficient algorithm.

The solution proposed in [16], whose framework we follow here, first constructs a supergraph $D_3$ of $D_2$ whose size is $\mathcal{O}(N)$. Due to the linear size of $D_3$, we can afford to compute the red graph of $D_3$ explicitly and remove non-extremal rungs from $D_3$. Graph $D_3$ is constructed by identifying the potential top, bottom, left, and right rungs of each ladder in $D_1$ and computing the edge set of $D_3$ as the union of the set of these rungs, the set of all edges in $D_1$ incident to points in $S'$, and the set of obstacle edges. In this section, we describe the I/O-efficient construction of subdivisions $D_3$ and $D_2$, once the set of rungs of subdivision $D_3$ has been computed. An I/O-efficient algorithm for computing the rungs is described in Section 14.2.4.

#### 14.2.3.1 Computing Subdivision $D_3$

Let $E'_R$ be the set of potential top, bottom, left, and right rungs. Then subdivision $D_3$ is defined by all obstacles edges (i.e., the set of edges in $E$), the set of rungs in $E'_R$, and the set $E'_S$ of *short edges* of $D_1$. A short edge is an edge which is completely contained in an edge of $D'$ and has at least one endpoint in $S'$.

Since sets $E$ and $E'_R$ are already given, we have to compute the set $E'_S$ of short edges and extract a representation of subdivision $D_3$ as an embedded planar graph from the edge set $E \cup E'_R \cup E'_S$ defining subdivision $D_3$. We describe these two steps next.

**Computing short edges.** The set $E'_S$ of short edges can be computed by answering ray-shooting queries on $P$ and shortening the edges in $E'$ appropriately. In particular, let $p \in S'$ be a vertex of $D'$. Then there are at most four edges in $E'$ incident to $p$.

For each such edge $E'$, we shoot a ray $\rho$ from $p$ in the direction of edge $e'$ until it hits an obstacle edge $e$. Let $q$ be the intersection point of $e$ and $\rho$, and let $e'' = (p, q)$. If $e'' \subseteq e'$, we add $e''$ to the set $E'_S$ of short edges. Otherwise, we add edge $e'$ to this set.

The ray-shooting queries used to compute edges $e''$ are axes-parallel, and there are $\mathcal{O}(N)$ of them to be answered. Hence, we use the endpoint dominance algorithm of [15] to answer these queries in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ I/Os. Given the set of answers to these queries, a single scan of this set is sufficient to decide for every edge $e' \in E'$ whether to add edge $e'$ or $e''$ to $E'_S$.

**Computing graph $D_3$.** A representation of subdivision $D_3$ as an unordered edge set $E \cup E'_R \cup E'_S$ is not very useful for computing $D_2$ from $D_3$. In particular, a representation of $D_3$ as vertex and edge sets is required. The geometric information pertaining to the vertices and edges of $D_3$ then provides us with a planar embedding $\hat{D}_3$ of graph $D_3$.

The vertex set of $D_3$ is easily constructed as the set of endpoints of all edges in $E \cup E'_R \cup E'_S$. We apply procedure DuplicateRemoval to remove duplicates from the generated vertex set. To compute the edge set of $D_3$, every obstacle edge has to be split at the endpoints of rungs and short edges lying on this obstacle edge. Once all obstacle edges have been split, duplicate rungs and short edges have to be removed from the edge set, since these edges may have been generated more than once.

The computation of rungs and short edges can easily be augmented so that every edge in $E'_R \cup E'_S$ stores the identities of the obstacle edges containing its endpoints. Hence, we can split the obstacle edges as follows: We sort the set of endpoints of edges in $E'_R \cup E'_S$ by the obstacle edges containing them and so that the endpoints lying on the same obstacle edge are sorted by their distances from one endpoint of that edge. We scan this sorted list of endpoints and the list $E$ of obstacles edges to split obstacle edges.

To remove duplicate rungs and short edges from the edge set of $D_3$, we apply operation DuplicateRemoval to this edge set.

In the above construction, we apply procedure DUPLICATEREMOVAL twice and sort and scan sets of size $\mathcal{O}(N)$ a constant number of times. Hence, the construction of graph $D_3$ from the edge set $E \cup E'_R \cup E'_S$ takes $\mathcal{O}(\text{sort}(N))$ I/Os. Together with the fact that the set of short edges can be computed in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ I/Os, we thus obtain the following lemma.

**Lemma 14.10** *Given the set $E$ of obstacle edges, the set $E'_R$ of rungs of subdivision $D_3$, and subdivision $D'$, a representation of $D_3$ as an embedded planar graph can be computed in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ I/Os using $\mathcal{O}\left(\frac{N}{B}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ blocks of external memory.*

#### 14.2.3.2 Computing Subdivision $D_2$

To construct subdivision $D_2$ from graph $D_3$, we compute the dual $D_3^*$ of graph $D_3$, extract the red graph of $D_3$ from $D_3^*$, and remove all edges dual to edges in the red graph from $D_3$. The result is a graph $D'_3$ whose faces define the regions of $D_2$. We scan the list of edges on the boundary of each face of $D'_3$ and merge consecutive collinear edges to produce a constant size description of each region in $D_2$.

Given a planar embedding $\hat{D}_3$ of $D_3$, it is shown in [100] how to compute the dual $D_3^*$ of $D_3$ in $\mathcal{O}(\text{sort}(N))$ I/Os. The computation of this algorithm can easily be augmented to color every vertex of $D_3^*$ as either red or blue, depending on the type of its corresponding region, and to mark every edge of $D_3^*$ as being dual to an obstacle edge or an edge of $D'$. Now we use procedure COPYVERTEXLABELS to identify all edges of $D_3^*$ which have at least one blue endpoint. Then we scan the edge set of $D_3^*$ to remove all edges with at least one blue endpoint and edges which are dual to obstacle edges. Let $E'_3$ be the resulting set of edges. The edges in $E'_3$ are the edges of the red graph of $D_3$. Each edge in $E'_3$ is dual to a non-extremal rung of a ladder in $D_3$. We scan set $E'_3$ to construct the set $E''_3$ of rungs dual to the edges in $E'_3$. Finally, we apply procedure SETDIFFERENCE to the edge set $E_3$ of $D_3$ and set $E''_3$ to remove the rungs in $E''_3$ from $D_3$. This produces graph $D'_3$.

The faces of graph $D_3'$ correspond to the regions of subdivision $D_2$. To construct a representation of $D_2$ as a collection of regions, we compute a representation of the faces of $D_3'$ as a collection of edge lists, each storing the edges of one face, clockwise around that face. Such a representation can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, as shown in [100]. Once this representation is given, a single scan of these edge lists suffices to compute constant size representations of regions $R(f)$, for all faces $f$ of $D_3'$. In particular, we scan each edge list and merge consecutive collinear edges of the same type (subsegment of an edge in $E$ or $E'$). By the discussion in Section 14.2.1, only $\mathcal{O}(1)$ edges remain per face.

In the above construction of subdivision $D_2$ from graph $D_3$, we apply procedure CopyVertexLabels and SetDifference, two $\mathcal{O}(\text{sort}(N))$ I/O procedures for constructing the dual $D_3^*$ of $D_3$ and extracting the edges on the boundary of each face of graph $D_3'$, and otherwise sort and scan lists of size $\mathcal{O}(N)$ a constant number of times. This takes $\mathcal{O}(\text{sort}(N))$ I/Os. Together with Lemmas 14.10 and 14.12, this proves the following result.

**Lemma 14.11** *Given a set $P$ of polygonal obstacles in the plane, subdivision $D_2$ can be computed in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ I/Os using $\mathcal{O}\left(\frac{N}{B}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ blocks of external memory, where $N$ is the total number of obstacle vertices.*

### 14.2.4 Computing the Rungs

The construction of subdivision $D_2$ in Section 14.2.3 assumes that the set $E_R'$ of extremal rungs of subdivision $D_1$ is given. Computing this set of rungs is the most difficult part of the algorithm. The remainder of this section is dedicated to describing a procedure for computing set $E_R'$ in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ I/Os. In [16], a plane-sweep algorithm is used to compute each type of extremal rungs separately. We follow this approach, but simplify the sweep-line data structure so that the approach leads to an I/O-efficient algorithm.

Since top, bottom, left, and right rungs are computed using four similar plane sweeps, we only describe the computation of top rungs. Let $E_h'$ be the set of horizontal

**Procedure** FINDTOPRUNGS

**Input:** The set $E'_h$ of horizontal edges of subdivision $D'$ and the set $E$ of obstacle edges.
**Output:** A set $E'_R$ of size $\mathcal{O}(N)$ which contains the top rungs of all ladders in $D_1$.

1: $E'_R \leftarrow \emptyset$
2: $L \leftarrow \langle (a, b) \rangle$, where $a$ and $b$ are the left and right sides of square $C$.
   {$L$ is the list of intervals intersected by the sweep-line, sorted from left to right.}
   {$C$ is the square containing all polygon vertices.}
3: Mark interval $(a, b)$ as "non-ladder".
4: $Y = \{y(p) : p \in S\} \cup \{y(e) : e \in E'_h\}$
   {$Y$ is the set of $y$-coordinates where the set of intervals in $L$ changes.}
5: **for** all $y \in Y$, sorted from $-\infty$ to $+\infty$ **do**
6:  **if** $y = y(p)$, for some point $p \in S$ **then**
7:  Let $e_1$ and $e_2$ be the two edges incident to $p$, where $e_1$ is to the left of or below $e_2$.
8:  **if** $p$ is the bottom corner of an obstacle in $P$ **then**
9:  PROCESSBOTTOMCORNER$(p, e_1, e_2, L, E'_R)$
10:  **else if** $p$ is the top corner of an obstacle in $P$ **then**
11:  PROCESSTOPCORNER$(p, e_1, e_2, L, E'_R)$
12:  **else**
13:  PROCESSNONEXTREMALCORNER$(p, e_1, e_2, L, E'_R)$
14:  **end if**
15:  **else**
16:  Let $e \in E'_h$ so that $y = y(e)$.
17:  PROCESSHORIZONTALEDGE$(e, L, E'_R)$
18:  **end if**
19: **end for**
20: **for** all remaining intervals $(l, r) \in L$ **do**
21:  Add the top-rung of interval $(l, r)$ to $E'_R$ if $(l, r)$ is a ladder interval.
22: **end for**

**Algorithm 14.1**
Finding the top-rungs of all ladders in $D_1$.

**Procedure** PROCESSBOTTOMCORNER$(p, e_1, e_2, L, E'_R)$

1: Find the interval $(l, r) \in L$ containing point $p$.
2: **if** $(l, r)$ is a ladder interval **then**
3: Add the top rung of $(l, r)$ to set $E'_R$.
4: **end if**
5: Replace $(l, r)$ by three non-ladder intervals $(l, e_1)$, $(e_1, e_2)$, and $(e_2, r)$ in $L$.

**Procedure** PROCESSTOPCORNER$(p, e_1, e_2, L, E'_R)$

1: Add the top rungs of all ladder intervals in $\{(l, e_1), (e_1, e_2), (e_2, r)\}$ to $E'_R$.
2: Replace $(l, e_1)$, $(e_1, e_2)$, and $(e_2, r)$ by a single non-ladder interval $(l, r)$ in $L$.

**Procedure** PROCESSNONEXTREMALCORNER$(p, e_1, e_2, L, E'_R)$

1: Let $(l, e_1)$ and $(e_1, r)$ be the two intervals incident to edge $e_1$.
2: Add the top rungs of all ladder intervals in $\{(l, e_1), (e_1, r)\}$ to $E'_R$.
3: Replace intervals $(l, e_1)$ and $(e_1, r)$ by two non-ladder intervals $(l, e_2)$ and $(e_2, r)$, respectively.

**Procedure** PROCESSHORIZONTALEDGE$(e, L, E'_R)$

1: Locate intervals $(l_1, r_1)$ and $(l_k, r_k)$ in $L$ which contain the endpoints of edge $e$.
2: Let $(l_2, r_2), \ldots, (l_{k-1}, r_{k-1})$ be the intervals between $(l_1, r_1)$ and $(l_k, r_k)$ in $L$.
3: Add the top-rungs of all ladder intervals in $\{(l_1, r_1), (l_k, r_k)\}$ to $E'_R$.
4: Mark intervals $(l_1, r_1)$ and $(l_k, r_k)$ as "non-ladder".
5: Mark intervals $(l_2, r_2), \ldots, (l_{k-1}, r_{k-1})$ as "ladder" and make edge $e$ their top rung.

**Algorithm 14.2**
The four procedures used for processing event points in Algorithm 14.1.

edges of subdivision $D'$. Then we use Algorithm 14.1 to compute all potential top rungs. The algorithm employs a plane-sweep in $(+y)$-direction to carry out its task. During the sweep, it maintains a set of intervals defined by intersections between the sweep-line $\ell$ and obstacle edges. In particular, let $e_1, \ldots, e_k$ be the edges in $E$ intersected by the sweep line, sorted from left to right. Then the set of intervals currently stored for $\ell$ is $(e_1, e_2), (e_2, e_3), \ldots, (e_{k-1}, e_k)$. Each such interval $(e_i, e_{i+1})$ is classified as *ladder* or *non-ladder* depending on whether the algorithm has already

found a rung whose endpoints lie on $e_i$ and $e_{i+1}$. In particular, for an interval $I = (e_i, e_{i+1})$, let $h$ be the highest edge in $E'_h$ below $\ell$ and intersecting both $e_i$ and $e_{i+1}$. Interval $I$ is a non-ladder interval if $h$ does not exist or the quadrilateral defined by $\ell$, $e_i$, $e_{i+1}$, and $h$ contains a point from $S \cup S'$. Otherwise, $I$ is a ladder interval.

During the sweep, an interval can be created or destroyed only when the sweep passes the $y$-coordinate of an endpoint of an edge in $E$. The type of an interval can change only when the sweep passes the $y$-coordinate of an edge in $E'_h$. The algorithm deals with these different types of event points and maintains the list of intervals, their classification, and for ladder intervals their top rungs. It is easy to verify that the algorithm maintains the list of intervals and their classification correctly, so that the rule for reporting top rungs is correct. We have to present a data structure to represent the sweep-line status so that the list of intervals can be maintained I/O-efficiently.

The data structure has to support updates of the status of a sequence of consecutive ladder intervals at the cost of $o(1)$ changes to the data structure per interval in the sequence. To see why this is necessary, observe that an edge $e \in E'_h$ may intersect $\Omega(N)$ obstacle edges, so that the status of $\Omega(N)$ intervals has to be updated when the sweep passes edge $e$. As there are $\Theta(N)$ edges in $E'_h$, the number of required updates of the sweep-line data structure would be $\Omega(N^2)$ if each updated interval caused $\Omega(1)$ updates of the data structure. In order to achieve $o(1)$ data structure updates per modified interval, our data structure, just as the one presented in [16], exploits the following observation.

**Observation 14.1 (Arikati et al. [16])** *Two consecutive ladder intervals have the same top rung.*

The data structure proposed in [16] exploits this fact by representing intervals and their status in two separate binary search trees. The first tree, $T$, stores the current set of intervals sorted from left to right. The second tree, $T'$, represents the status of all intervals in a compressed form by storing only one entry for each consecutive sequence of intervals of the same type. Since every event point generates or destroys only

$\mathcal{O}(1)$ intervals, procedure FINDTOPRUNGS performs only $\mathcal{O}(N)$ updates of tree $T$. Counting the number of updates of tree $T'$ is not quite that easy. However, it can be shown that every generation or deletion of an entry in tree $T'$ can be charged to a blue region in subdivision $D_1$ so that every region is charged only $\mathcal{O}(1)$ times. Every change of the status of an existing entry in $T'$ can be charged to an edge in $E'_h$. Hence, only $\mathcal{O}(N)$ updates of tree $T'$ are performed, and procedure FINDTOPRUNGS takes $\mathcal{O}(N \log N)$ time.

In order to obtain a data structure which allows the sweep to be performed in $\mathcal{O}(\text{sort}(N))$ I/Os, we could try to replace trees $T$ and $T'$ by buffer trees. Unfortunately, this simple idea cannot be applied, as the updates of tree $T$ are driven by the answers to queries on tree $T'$ and vice versa, so that immediate query responses are required. The efficiency of the buffer tree, however, is achieved by *delaying* query responses and answering queries whenever a large enough number of queries has accumulated.

Next we show how to represent the sweep-line status using only a single buffer tree. The fact that the buffer tree delays the processing of updates and queries still creates problems that have to be dealt with. In order to support our claim that our data structure is simpler than the one proposed in [16], we present the algorithm as if it used an $(a, b)$-tree which does not buffer updates or queries. In Section 14.2.4.2, we discuss the problems created by delayed updates and show how to deal with them.

#### 14.2.4.1 A Simplified Sweep-Line Data Structure

Our sweep-line data structure consists of a single $(a, b)$-tree $T$ [99]. The leaves of $T$ store the current set of intervals sorted from left to right. Every internal node stores the obstacle edges separating the intervals stored at descendants of its children.

In order to obtain a classification of the intervals as ladder and non-ladder intervals, every interval $I$ stores a label $\lambda(I) = (y, \tau, \rho)$, and every internal node $v$ of $T$ stores a label $\lambda(v) = (y, \tau, \rho)$. For an interval $I$, label $\lambda(I) = (y, \tau, \rho)$ signifies that interval $I$ was of type $\tau$ just after sweep-line $\ell$ crossed $y$-coordinate $y$. If $\tau =$ "ladder",

$\rho$ is the topmost rung of the ladder in interval $I$ below or on the horizontal line at $y$-coordinate $y$. For an internal node, label $\lambda(v) = (y, \tau, \rho)$ signifies that every interval $I$ stored at a descendant of $v$ was of type $\tau$ just after sweep-line $\ell$ crossed $y$-coordinate $y$. If $\tau =$ "ladder", $\rho$ is the topmost rung of the ladder in interval $I$ below or on the horizontal line at $y$-coordinate $y$.

We maintain the invariant that at any time, the correct type and top rung of an interval $I$ can be determined as follows: Let $v_0$ be the leaf of $T$ storing interval $I$, and let $v_1, \ldots, v_k$ be the proper ancestors of $v_0$ in $T$. Let $\lambda(I) = (y_0, \tau_0, \rho_0)$, and let $\lambda(v_i) = (y_i, \tau_i, \rho_i)$, for $1 \leq i \leq k$. Then interval $I$ is of type $\tau_i$ and its top rung is $\rho_i$ if $\tau_i =$ "ladder", where $y_i = \max\{y_j : 0 \leq j \leq k\}$. That is, in terms of the plane-sweep, the most recent information stored on the path $(v_0, \ldots, v_k)$ in $T$ is the correct characterization of interval $I$.

The basic query operation on tree $T$ is now operation REPORT($I$), which searches for interval $I$ and reports its top rung if interval $I$ is a ladder interval. To do this, it traverses the path in $T$ from the root to the leaf storing interval $I$ and finds the triple with maximal $y$-coordinate stored on this path. By the above invariant, this triple contains the correct type and top rung of interval $I$. Given at least one point in interval $I$, the obstacle edges stored at the internal nodes of $T$ are sufficient to locate the leaf storing interval $I$.

Next we discuss the required updates of tree $T$ when the sweep-line passes an event point. We discuss the update procedures in Algorithm 14.2 for the four different types of event points separately and argue that each of them maintains the above invariant.

PROCESSBOTTOMCORNER: A bottom corner $p \in S$ of an obstacle is a vertex whose incident edges $e_1$ and $e_2$ are both above $p$. Let $e_1$ be to the left of $e_2$. Then the top rung of the interval $I = (l, r)$ containing point $p$ has to be reported if interval $I$ is a ladder interval, and interval $I$ has to be replaced by three non-ladder intervals $(l, e_1)$, $(e_1, e_2)$, and $(e_2, r)$. To achieve this, we apply an operation REPORTANDSPLIT($I, e_1, e_2, y(p)$) to tree $T$. This operation first applies operation REPORT($I$) to locate interval $I$ and report its top rung if necessary.

Then it replaces interval $I$ with three new intervals $I_1 = (l, e_1)$, $I_2 = (e_1, e_2)$, and $I_3 = (e_2, r)$. Every interval $I_j$, $j \in \{1, 2, 3\}$, is assigned label $\lambda(I_j) =$ ($y(p)$, "non-ladder", **null**). Since $y < y(p)$, for every label $\lambda(v) = (y, \tau, \rho)$ stored at an ancestor of intervals $I_1$, $I_2$, and $I_3$ in $T$, this correctly marks all three intervals as non-ladder intervals.

The replacement of interval $I$ with intervals $I_1$, $I_2$, and $I_3$ may cause tree $T$ to become unbalanced. Tree $T$ can be rebalanced using the standard procedure for rebalancing an $(a, b)$-tree after an INSERT operation.

PROCESSTOPCORNER: A top corner $p \in S$ of an obstacle is a vertex whose incident edges $e_1$ and $e_2$ are both below $p$. Let $e_1$ be to the left of $e_2$. Then the top rungs of the ladder intervals among intervals $I_1 = (l, e_1)$, $I_2 = (e_1, e_2)$, and $I_3 = (e_2, r)$ incident to edges $e_1$ and $e_2$ need to be reported, and intervals $I_1$, $I_2$, and $I_3$ need to be replaced by a single non-ladder interval $I = (l, r)$. To achieve this, apply three operations $o_1 =$ REPORTANDREPLACE$(I_1, I, y(p))$, $o_2 =$ REPORTANDDELETE$(I_2)$, and $o_3 =$ REPORTANDDELETE$(I_3)$ to tree $T$. Operation $o_1$ searches for interval $I_1$, reports its top rung if it is a ladder interval, and replaces it by interval $I$. The label of interval $I$ is set to $\lambda(I) =$ ($y(p)$, "non-ladder", **null**), thereby correctly marking interval $I$ as a non-ladder interval. Operations $o_2$ and $o_3$ search for intervals $I_2$ and $I_3$, report their top rungs if necessary and delete these intervals. After deleting intervals $I_2$ and $I_3$, tree $T$ can be rebalanced using the standard rebalancing procedure for DELETE operations on $(a, b)$-trees.

Note that procedure PROCESSTOPCORNER is supplied only with the two edges $e_1$ and $e_2$ incident to point $p$. This allows intervals $I_1$, $I_2$, and $I_3$ to be located in $T$, but is not sufficient to provide operation $o_1$ with a complete description of interval $I$. In order to work around this problem, we apply operations $o_2$ and $o_3$ first, and augment operation $o_3$ so that it reports the right boundary edge of interval $I_3$, which is also the right boundary of interval $I$. Together with the left

boundary of interval $I_1$, operation $o_1$ can now compute a complete description of interval $I$.

Another issue is that the search information in $T$ may be invalidated by operation $o_1$. This happens if the separating obstacle edges to the right of the path traversed by operation $o_1$ intersect interval $I$ because interval $I$ spans three of the intervals previously stored in $T$. Operation $o_1$ can change these edges to the right boundary of interval $I$ while it traverses the path in $T$ to the leaf storing interval $I_1$.

ProcessNonExtremalCorner: A non-extremal corner $p \in S$ of an obstacle is a vertex so that an obstacle edge $e_1$ is incident to $p$ from below, and another obstacle edge $e_2$ is incident to $p$ from above. At such an event point, the following updates are necessary. Let $I_1 = (l, e_1)$ and $I_2 = (e_1, r)$ be the two intervals on both sides of edge $e_1$. Then the top rungs of both intervals need to be reported, depending on whether they are ladder intervals, and intervals $I_1$ and $I_2$ need to be replaced by two non-ladder intervals $I_1' = (l, e_2)$ and $I_2' = (e_2, r)$, respectively. This can be achieved by applying two operations ReportAndReplace$(I_1, I_1', y(p))$ and ReportAndReplace$(I_2, I_2', y(p))$ to tree $T$.

ProcessHorizontalEdge: The last type of event point is the $y$-coordinate of a segment $s \in E_h'$. Let $p_l$ and $p_r$ be the left and right endpoints of segment $s$, and let $I_l$ and $I_r$ be the intervals containing these two endpoints, respectively. Then the top rungs of intervals $I_l$ and $I_r$ need to be reported, depending on whether these intervals are ladder intervals, intervals $I_l$ and $I_r$ have to be marked as non-ladder intervals, and all intervals between $I_l$ and $I_r$ have to be marked as ladder intervals with top rung $s$. We achieve this by applying an operation $o =$ MakeLadder$(I_l, I_r, y(s))$ to tree $T$. Operation $o$ searches down the tree until it finds the first node $v$ so that intervals $I_l$ and $I_r$ are stored at descendants of different children of $v$. Let $w_1, \ldots, w_k$ be the children of $v$,

and let $w_i$ and $w_j$ be the two children so that intervals $I_l$ and $I_r$ are stored at descendants of $w_i$ and $w_j$, respectively. Then operation $o$ changes the label of every node $w_h$, $i < h < j$, to $\lambda(w_h) = (y(s), \text{“ladder”}, s)$. Now we split operation $o$ into two operations $o_1 = \textsc{MakeRightLadder}(I_l, y(s))$ and $o_2 = \textsc{MakeLeftLadder}(I_r, y(s))$. Operation $o_1$ continues down the path from $w_i$ to the leaf storing interval $I_l$. Operation $o_2$ follows the path from $w_j$ to the leaf storing interval $I_r$. At every visited internal node $v'$, operation $o_1$ performs the following operation: Let $w'_1, \ldots, w'_k$ be the children of node $v'$, and let interval $I_l$ be stored at a descendant of $w'_i$. Then the label of every node $w'_h$, $i < h \leq k$, is changed to $\lambda(w'_h) = (y(s), \text{“ladder”}, s)$. When the leaf $v_0$ storing interval $I_l$ is reached, the top rung of interval $I$ is reported if necessary, based on the information collected by operations $o$ and $o_1$ along the path from the root of $T$ to leaf $v_0$. The label of interval $I_l$ is set to $\lambda(I_l) = (y(s), \text{“non-ladder”}, \textbf{null})$. Operation $o_2$ performs the same updates w.r.t. interval $I_r$, but labels all intervals to the left of the search path as ladder intervals. It is easily verified that an interval appears between $I_l$ and $I_r$ if and only if it is stored at a descendant of a node $w_h$ or $w'_h$ in $T$ whose label is changed to “ladder”. Hence, this procedure correctly updates the type information of every interval.

Each of the above update procedures for the four different types of event points traverses a constant number of root-to-leaf paths in $T$. Each procedure spends $\mathcal{O}(b)$ time per visited node. In internal memory, we would choose $b = \mathcal{O}(1)$, so that every event point can be processed in $\mathcal{O}(\log N)$ time. As there are $\mathcal{O}(N)$ event points, procedure FindTopRungs finds all top rungs in $\mathcal{O}(N \log N)$ time.

#### 14.2.4.2 Buffering Updates

In order to make the sweep-line data structure I/O-efficient, the first step is to turn the $(a, b)$-tree into a buffer tree. Query and update procedures remain the same. But they are processed in a batched fashion, so that processing $\mathcal{O}(N)$ queries and

updates now takes $\mathcal{O}(\text{sort}(N))$ I/Os. However, the delayed processing of updates creates problems:

(i) The $x$-order of obstacle edges and hence the order of intervals defined by these edges is not a total order. While all segments intersected by a horizontal line have a well-defined order, edges whose $y$-spans are disjoint are incomparable. This does not create any problems for a standard $(a, b)$-tree, as queries and updates are processed immediately, and the search information in tree $T$ can be updated so that all separating edges stored at internal nodes of $T$ intersect the current sweep-line. In a buffer tree, on the other hand, delayed updates can lead to the situation that some nodes in $T$ store splitter edges which are incomparable to points on the current sweep-line because they are completely below the sweep-line. Thus, it is not clear at this point at which child of such a node to continue the search for a particular interval.

(ii) We argued in Section 14.2.4.1 that when operation ReportAndReplace is applied in procedure ProcessTopCorner, the information about the right boundary of interval $I$ can be collected by first deleting interval $I_3$ and then replacing interval $I_1$ by interval $I$. In a buffer tree, this strategy cannot be applied because the deletion of interval $I_3$ would have to be processed immediately. This makes it impossible to guarantee that at every node of $T$ a large enough number of queries and updates has accumulated before emptying the buffer of this node. But this is crucial for the I/O-efficiency of the buffer tree.

Our solution for Problem (i) is to define a total order of the intervals by precomputing the set of all intervals defined by obstacle edges and numbering them in a manner consistent with the partial $x$-order of these intervals. Intervals are then stored in the buffer tree sorted according to their numbers. This solves the first problem, but creates a new problem:

(iii) An update operation cannot search for an interval in $T$ using the location of a point $p$ in the interval, since tree $T$ stores no geometric search information.

Thus, every update information has to be provided with the numbers of the intervals involved in the update.

The fourth and final problem is the following:

(iv) The procedure for updating splitter values that are invalidated by an application of operation ReportAndReplace requires the immediate processing of this operation, which is not feasible using a buffer tree. Hence, the numbering of the intervals has to be defined so that if the splitter values in the buffer tree are chosen carefully, an application of operation ReportAndReplace does not invalidate any splitters.

As mentioned above, our solution for Problem (i) is to precompute the set of intervals and assign a unique number $\nu(I)$ to each interval $I$. We obtain a total order of the set of intervals as the total order defined by numbering $\nu$. Below we discuss the computation of this numbering and argue that it satisfies the condition in Problem (iv). To solve Problems (ii) and (iii), we compute three lists $L_R$, $L_D$, and $L_H$. List $L_R$ stores pairs $(I_1, I_2)$, where interval $I_1$ is replaced by interval $I_2$ during the sweep. List $L_D$ stores single intervals that are removed from tree $T$ using operation ReportAndDelete during the sweep. List $L_H$ stores pairs $(I_l, I_r)$ of intervals, where intervals $I_l$ and $I_r$ contain the left and right endpoints of a segment in $E'_h$. The elements of lists $L_R$, $L_D$, and $L_H$ are sorted by the $y$-coordinates of the corresponding event points. Every interval $I$ in lists $L_R$, $L_D$, and $L_H$ stores both its bounding segments as well as its number $\nu(I)$. The I/O-efficient version of Algorithm 14.1 now proceeds as follows:

We use a buffer tree instead of an $(a, b)$-tree to store the set of intervals intersected by the sweep-line. These intervals are sorted by their numbers, which is consistent with the $x$-order of these intervals, by the definition of numbering $\nu$. Operation ReportAndReplace retrieves the next pair $(I_1, I_2)$ from list $L_R$. It searches for interval $I_1$ in $T$, using number $\nu(I_1)$, and replaces $I_1$ with $I_2$. Operation ReportAndSplit retrieves the next three pairs $(I, I_1)$, $(I, I_2)$, and $(I, I_3)$ from $L_R$. It searches for interval $I$ in $T$ and replaces this interval with intervals $I_1$, $I_2$, and $I_3$. Operation ReportAndDelete retrieves the next interval $I$ from list $L_D$, searches

for interval $I$ in $T$, and removes this interval. Operation MAKELADDER retrieves the next pair $(I_l, I_r)$ from list $L_H$ and makes a ladder between intervals $I_l$ and $I_r$.

The correctness of this modified version of Algorithm 14.1 is easily verified. Its I/O-complexity is $\mathcal{O}(\text{sort}(N))$ because it performs $\mathcal{O}(N)$ updates and queries on a buffer tree $T$ of size $\mathcal{O}(N)$ and scans lists $L_R$, $L_D$, and $L_H$. Below we show how to compute numbering $\nu$ as well as lists $L_R$, $L_D$, and $L_H$ in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ I/Os, which proves the following lemma.

**Lemma 14.12** *Given a set $P$ of polygonal obstacles and subdivision $D'$ defined by the set $S$ of obstacle vertices, the set of top rungs of subdivision $D_3$ can be computed in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ I/Os using $\mathcal{O}\left(\frac{N}{B}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ blocks of external memory, where $N = |S|$.*

**The replacement tree.** To compute numbering $\nu$ and lists $L_R$, $L_D$, and $L_H$, we use a rooted tree defined on the set of all intervals. We call this tree the *replacement tree* $T_R$. We show that this tree can be computed in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ I/Os. The extraction of numbering $\nu$ and lists $L_R$, $L_D$, and $L_H$ from tree $T_R$ takes $\mathcal{O}(\text{sort}(N))$ I/Os, as shown below.

To construct the vertex set of $T_R$, we have to compute the set of all intervals defined by edges of the obstacles in $P$. This set of intervals is easily obtained from a trapezoidation of the obstacle set (see Figure 14.7). In particular, the $y$-span of each trapezoid defines the life span of the interval defined by the two obstacle edges on its boundary during the sweep. The interval is created when the sweep passes the bottom boundary of the trapezoid, and it is replaced by a new interval when the sweep passes the top boundary of the trapezoid. Since a trapezoid is an interval augmented with the range of $y$-coordinates where this interval is valid, we henceforth consider intervals and trapezoids to be the same, choosing between these two names for the same object depending on the context.

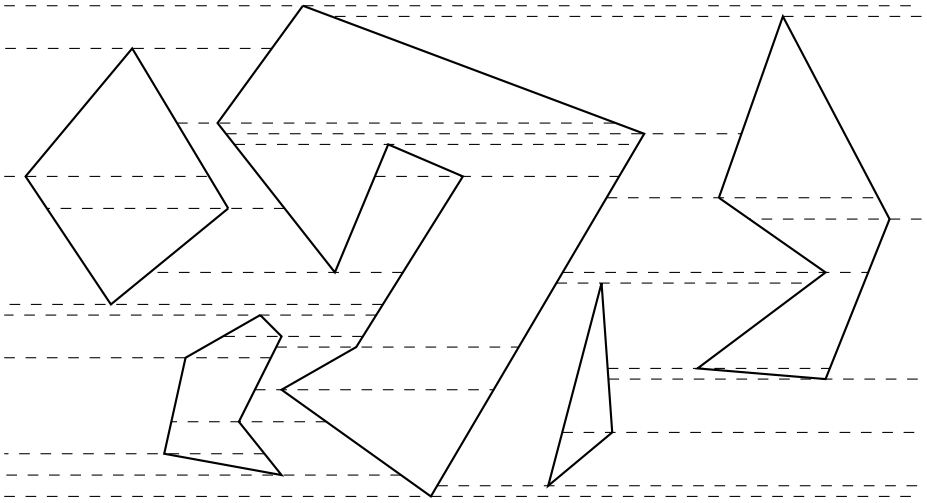


**Figure 14.7**
A trapezoidation of a set of polygons.

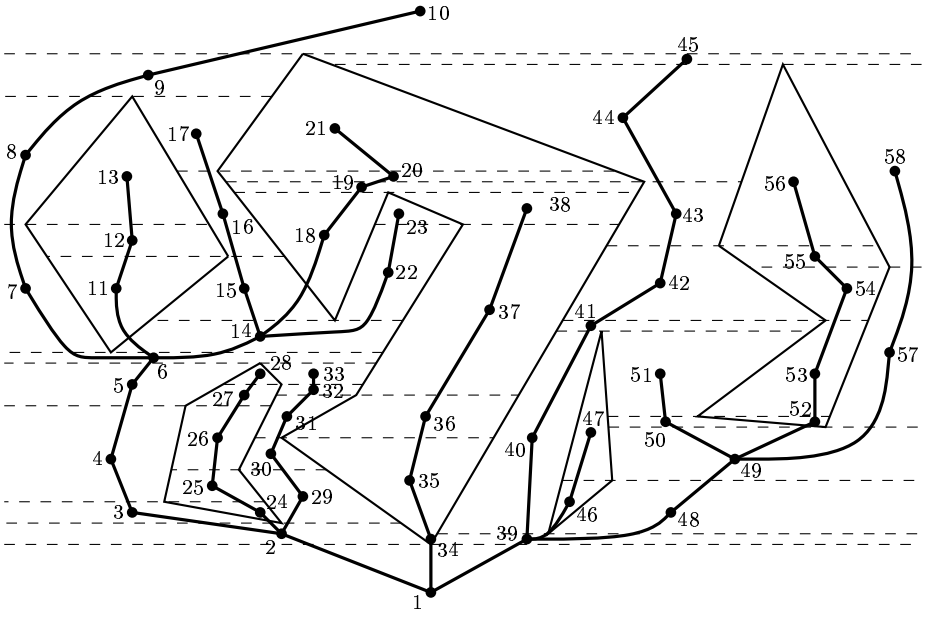



**Figure 14.8**
The replacement tree of the above trapezoidation. A preorder numbering of this tree defines a total order of the trapezoids with a particularly nice property.

As already mentioned, tree $T_R$ contains one node per trapezoid. The root of $T_R$ is the unbounded trapezoid extending to infinity in $(-y)$-direction. Every other trapezoid has a parent defined as follows: Consider an obstacle vertex $p \in S$. If $p$ is a bottom corner, the three trapezoids $I_1$, $I_2$, and $I_3$ incident to $p$ from above are the children of the trapezoid $I$ below $p$. If $p$ is a top corner, let $I_1$, $I_2$, and $I_3$ be the three trapezoids incident to $p$ from below, sorted from left to right. Then the trapezoid $I$ above $p$ is the child of trapezoid $I_1$. Finally, for a non-extremal corner, there are two trapezoids $I_1$ and $I_2$ incident to $p$ from below and two trapezoids $I_1'$ and $I_2'$ incident to $p$ from above. Let $I_1$ be to the left of $I_2$, and $I_1'$ be to the left of $I_2'$. Then $I_1'$ is the child of $I_1$, and $I_2'$ is the child of $I_2$. This construction is illustrated in Figure 14.8.

**Lemma 14.13** *The replacement tree $T_R$ of the set of intervals defined by the obstacles in $P$ can be computed in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ I/Os and $\mathcal{O}\left(\frac{N}{B}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ blocks of external memory.*

*Proof.* A trapezoidation of obstacle set $P$ can be computed in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ I/Os using the endpoint dominance algorithm of [15]. In particular, this algorithm computes the set of horizontal edges incident to every obstacle vertex and the obstacle edges containing the other endpoints of these edges. That is, after applying this algorithm, we obtain a representation of the trapezoidation as the set of obstacle edges and horizontal edges defining the trapezoidation. Using the same algorithm as for deriving a representation of subdivision $D_3$ as an embedded planar graph (see page 302), we can derive a representation of the trapezoidation as an embedded planar graph in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.

Using an algorithm of [100], the faces of the trapezoidation can be identified in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. The output of the algorithm is a representation of the faces as a collection of lists, each storing the vertices of one face clockwise around the face. From this representation we generate pairs $(v, f)$, where $v$ is a vertex on the boundary of face $f$. Then we sort the list of these pairs lexicographically, so that for each vertex $v$, pairs $(v, f_1), \ldots, (v, f_4)$ are stored consecutively. The edge set of tree $T_R$ can now be produced in a single scan of this list of pairs. □

**Computing a total order of the intervals.** In order to obtain a total order of all trapezoids, we compute a preorder numbering $\nu$ of $T_R$ which is consistent with a depth-first traversal of $T_R$ that visits the children of every node in left-to-right order (see Figure 14.8). Then $I_1 < I_2$ if $\nu(I_1) < \nu(I_2)$. The following lemma is easy to show by induction on the set of obstacle vertices, which is where the set of intervals intersected by the sweep-line changes.

**Lemma 14.14** *Let $I_1, \ldots, I_k$ be the set of trapezoids intersected by a horizontal line $\ell$, sorted from left to right. Then $\nu(I_1) < \cdots < \nu(I_k)$.*

Lemma 14.14 and the above discussion establish that the total order defined by preorder numbering $\nu$ can be used to sort the intervals in $T$ from left to right.

Next we describe a rule for choosing the splitters in tree $T$ so that ReportAndReplace operations do not invalidate the splitters in $T$. Let $w_i$ and $w_{i+1}$ be two consecutive children of a node $v$ in $T$. Then we choose the splitter between $w_i$ and $w_{i+1}$ to be $\sigma_i = \min\{\nu(I) : I \in \mathcal{I}(w_{i+1})\}$, where $\mathcal{I}(x)$ is the set of intervals stored in the subtree rooted at node $x$. Note that this is a rule we apply whenever we choose a new splitter. It is not an invariant we maintain at all times.

Now consider an application of operation ReportAndReplace or ReportAndSplit which replaces an interval $I_1$ with another interval $I_2$. Let $v_0$ be the leaf storing interval $I_1$, and let $v_1, \ldots, v_k$ be the ancestors of leaf $v_0$, sorted by increasing distance from $v_0$. That is, $v_k$ is the root of tree $T$. For $1 \leq i \leq k$, let $\sigma_i$ and $\sigma_i'$ be the two splitters stored at node $v_i$ so that for all intervals $I' \in \mathcal{I}(v_{i+1})$, $\sigma_i \leq \nu(I') < \sigma_i'$. Then we prove the following lemma.

**Lemma 14.15** *If operation* ReportAndReplace *or* ReportAndSplit *replaces an interval $I_1$ with an interval $I_2$, and $v_1, \ldots, v_k$ are the ancestors of the leaf $v_0$ storing interval $I_1$, then $\sigma_i \leq \nu(I_2) < \sigma_i'$, for all $1 \leq i \leq k$.*

*Proof.* Since we assume that the search information stored in tree $T$ is correct before the replacement of interval $I_1$ with interval $I_2$, $\sigma_i \leq \nu(I_1) < \sigma_i'$, for all $1 \leq i \leq k$. This immediately implies that $\sigma_i < \nu(I_2)$, for all $1 \leq i \leq k$, because interval $I_2$ is a

descendant of interval $I_1$ in tree $T_r$ and hence $\nu(I_1) < \nu(I_2)$. We have to show that $\nu(I_2) < \sigma'_i$, for all $1 \leq i \leq k$.

For the sake of this proof, we introduce the notion of a *slot* in tree $T$. Every slot holds an interval $I_2$. Operation ReportAndReplace leaves the set of slots unchanged, but changes the content of the slot holding interval $I_1$ by storing interval $I_2$ in this slot. Operation ReportAndDelete destroys the slot holding the deleted interval. Operation ReportAndSplit creates two new slots holding intervals $I_2$ and $I_3$. The two create slots are immediately to the right of the slot holding interval $I_1$.

Now consider a splitter $\sigma'_i$. Then $\sigma'_i = \nu(I')$, for some interval $I'$. Let $\ell$ be a horizontal line intersecting interval $I'$, and let $I_0$ be the ancestor of interval $I_1$ in $T_R$ which is intersected by line $\ell$. If we can show that interval $I_0$ is to the left of interval $I'$, we obtain that $\nu(I_0) < \nu(I')$, by Lemma 14.14. Since $I_0$ and $I'$ are intersected by the same horizontal line, $I'$ is not a descendant of $I_0$. Interval $I_2$ is a descendant of interval $I_0$. Hence, $\nu(I_2) < \nu(I') = \sigma'_i$, by the property of any preorder numbering that the nodes in any subtree of $T_R$ rooted at some node $I_0$ are numbered consecutively. In order to show that interval $I_0$ is to the left of interval $I'$, we show that the slot holding interval $I_0$ is to the left of the slot holding interval $I'$.

So assume the contrary, i.e., that $I_0 = I'$, or interval $I_0$ is stored in a slot to the right of the slot storing interval $I'$. In both cases, the slot holding interval $I_0$ is stored at a descendant of a child of node $v_i$ which is to the right of splitter $\sigma'_i$. Hence, if $I_0$ and $I_1$ are stored in the same slot, we obtain the desired contradiction. Otherwise, the slot holding interval $I_1$ has been created from the slot holding interval $I_0$ by a sequence of ReportAndSplit operations. This operation changes the content of an existing slot and creates two new slots to the right of the existing slot. Hence, the slot holding interval $I_1$ is to the right of splitter $\sigma'_i$ in this case as well. □

By Lemma 14.15, no updates of the splitters in the tree are required after an application of operation ReportAndReplace or ReportAndSplit. Hence, the plane-sweep can be carried out I/O-efficiently using a buffer tree instead of an $(a, b)$-tree, provided that the intervals in $T$ are sorted according to the preorder numbering $\nu$

defined above and splitters are computed as described above. We have to show that preorder numbering $\nu$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.

This can be done using the Euler tour technique and list-ranking. The Euler tour describes a depth-first traversal of tree $T_R$. We have to ensure that it visits the children of every node in left-to-right order. For every vertex $I \in T_R$ with incident edges $(I, I_1), \ldots, (I, I_k)$, let $I_1$ be the parent of $I$ in $T_R$, and let $I_2, \ldots, I_k$ be the children of $I_1$, sorted from left to right. Then we define $\text{succ}((I_j, I)) = (I, I_{j+1})$, for $1 \leq j < k$, and $\text{succ}((I_k, I)) = (I, I_1)$. This produces the desired Euler tour.

**Computing the replacement and deletion lists.** Lists $L_R$ and $L_D$ can be computed in a natural manner from tree $T_R$. In particular, every edge $(I_1, I_2)$ in $T_R$ defines an entry in list $L_R$ because there is an edge $(I_1, I_2)$ in $T_R$ if and only if interval $I_1$ is replaced by interval $I_2$ during the sweep. Similarly, every leaf of $T_R$ is removed from $T$ during the sweep, using operation ReportAndDelete. Hence, the contents of lists $L_R$ and $L_D$ can be computed in a single scan of the vertex and edge sets of tree $T_R$. To arrange the elements $(I_1, I_1'), \ldots, (I_s, I_s')$ in $L_R$ in the correct order, we sort them by the $y$-coordinates of the edges shared by trapezoids $I_j$ and $I_j'$, $1 \leq j \leq s$. We sort the elements $I_1, \ldots, I_t$ of list $L_D$ by the $y$-coordinates of their top edges. This computation of lists $L_R$ and $L_D$ clearly takes $\mathcal{O}(\text{sort}(N))$ I/Os.

**Computing list $L_H$.** To construct list $L_H$, we have to compute the intervals $I_l$ and $I_r$ containing the endpoints of each edge $e \in E_h'$. That is, we have to answer $\mathcal{O}(N)$ point location queries on the trapezoidation of obstacle set $P$. To do this, we scan the vertex set of tree $T_R$ and add for each node $I \in T_R$, the left boundary of trapezoid $I$ to a list $X$. Then a point $p$ is contained in trapezoid $I$ if and only if a ray shot from $p$ in $(-x)$-direction hits this left boundary of interval $I$. Hence, we can answer the point location queries for all segment endpoints in $\mathcal{O}\left(\frac{N}{DB} \log_{\frac{M}{DB}} \frac{N}{DB}\right)$ I/Os by applying the endpoint dominance algorithm of [15] to the set $X$ of left boundaries and the set of endpoints of all segments in $E_h'$. The answers to all queries can be now combined to

pairs $(I_l, I_r)$ in $\mathcal{O}(\text{sort}(N))$ I/Os. Given the set of these pairs, we obtain list $L_H$ by sorting all pairs $(I_l, I_r)$ by the $y$-coordinates of their corresponding edges in $E'_h$.

We summarize the above discussion in the following lemma, which completes the proof of Lemma 14.12.

**Lemma 14.16** *A total order of all intervals defined by the edges of all obstacles in $P$ as well as lists $L_R$, $L_D$, and $L_H$ can be computed in $\mathcal{O}\left(\frac{N}{DB}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ I/Os using $\mathcal{O}\left(\frac{N}{B}\log_{\frac{M}{DB}}\frac{N}{DB}\right)$ blocks of external memory.*

## 14.3 A Planar $L_2$-Spanner

Given that planar $L_1$-Steiner spanners for sets of points and obstacles in the plane can be computed I/O-efficiently, we can use them to compute planar $L_2$-Steiner spanners. For the sake of brevity, we present proofs only for the case of polygonal obstacles, since the case of a point set is simpler. The construction is based on the following observation: If an edge $e$ is close to vertical, i.e., its angle to the $y$-axis is at most $\theta$, then the $L_1$-length of $e$ is a $(\cos\theta + \sin\theta)$-approximation of the Euclidean length of edge $e$. For simplicity, we only use that $\|e\|_1 \leq (1+\sin\theta)\|e\|_2$. Hence, we construct an $L_2$-spanner with spanning ratio $1+\varepsilon$ for a set $P$ of polygonal obstacles as follows: We choose a constant $\varepsilon'$ so that $(1+\varepsilon')^2 \leq 1+\varepsilon$, e.g., $\varepsilon' = \varepsilon/3$. Let $0 < \theta < \pi/2$ so that $\sin\theta = \varepsilon'$. Then we choose $\pi/\theta = \mathcal{O}(1/\varepsilon)$ coordinate systems at angles $0, \theta, 2\theta, \ldots$ to a fixed reference coordinate system and construct $L_1$-Steiner spanners $G_0, \ldots, G_q$ for $P$ with stretch factor $1+\varepsilon'$ in these coordinate systems. Let $G$ be the graph obtained as the superimposition of graphs $G_0, \ldots, G_q$. That is, the vertex set of graph $G$ contains all vertices of graphs $G_0, \ldots, G_q$ as well as all intersection points between edges in these graphs. The edge set contains all edges obtained by splitting the edges of graphs $G_0, \ldots, G_q$ at their intersection points. The following lemma shows that graph $G$ is a Euclidean Steiner spanner with spanning ratio $1+\varepsilon$ for obstacle set $P$.

**Lemma 14.17 (Arikati et al. [16])** *Graph $G$ is an $L_2$-spanner with spanning ratio $1+\varepsilon$.*

*Proof.* Consider the shortest path $Q = (p = x_0, x_1, \ldots, x_k = q)$ in the $L_2$-metric between two points $p$ and $q$. For each edge $(x_i, x_{i+1})$, we construct a spanner path $Q'_i$ as follows: We find the coordinate system so that the angle between edge $(x_i, x_{i+1})$ and the $y$-axis is at most $\theta$. Let $G_j$ be the $L_1$-spanner constructed w.r.t. this coordinate system. Then let $Q_i$ be an $L_1$-spanner path of length $(1+\varepsilon')\|(x_i, x_{i+1})\|_1$ in $G_j$. Since $G$ is the superimposition of graphs $G_0, \ldots, G_q$, graph $G$ contains a spanner path $Q'_i$ of length

$$\begin{aligned}
\|Q'_i\|_2 &\leq \|Q'_i\|_1 \\
&= \|Q_i\|_1 \\
&\leq (1+\varepsilon')\|(x_i, x_{i+1})\|_1 \\
&\leq (1+\varepsilon')^2\|(x_i, x_{i+1})\|_2 \\
&\leq (1+\varepsilon)\|(x_i, x_{i+1})\|_2.
\end{aligned}$$

Hence, the path $Q' = Q'_0 \circ \cdots \circ Q'_{k-1}$ is a path in graph $G$ whose length is

$$\begin{aligned}
\|Q'\|_2 &= \sum_{j=0}^{k-1} \|Q'_j\| \\
&\leq (1+\varepsilon)\sum_{j=0}^{k-1} \|(x_i, x_{i+1})\|_2 \\
&= (1+\varepsilon)\|Q\|_2.
\end{aligned}$$

□

The following lemma shows that graph $G$ is small.

**Lemma 14.18 (Arikati et al. [16])** *Graph $G$ has size $\mathcal{O}(N/\varepsilon^4)$.*

*Proof.* Since graph $G$ is planar, it is sufficient to show that the number of vertices in $G$ is $\mathcal{O}(N/\varepsilon^4)$. The number of vertices in graphs $G_0, \ldots, G_q$ can be bounded by

$\mathcal{O}(N/\varepsilon^3)$ because $q = \mathcal{O}(1/\varepsilon)$ and $|G_i| = \mathcal{O}(N/\varepsilon^2)$, by Lemma 14.7. We show that for any pair of graphs $G_i$ and $G_j$, the number of intersections between edges in $G_i$ and $G_j$ is bounded by $\mathcal{O}(N/\varepsilon^2)$. As there are $\mathcal{O}(1/\varepsilon^2)$ such pairs of graphs, this gives the desired bound on the number of vertices in graph $G$.

In order to prove that the number of intersections between the edges of two graphs $G_i$ and $G_j$ is bounded by $\mathcal{O}(N/\varepsilon^2)$, let $D_i'$ and $D_j'$ be the two subdivisions of the plane into box and donut cells w.r.t. the coordinate systems used to define graphs $G_i$ and $G_j$. For each cell $R$ in $D_i'$, the edges of graph $G_i$ contained in $R$ are contained in $\mathcal{O}(1/\varepsilon)$ lines. The same is true for every cell in $D_j'$ w.r.t. graph $G_j$. Hence, for every pair of regions $R \in D_i'$ and $R' \in D_j'$, the number of intersections between edges of $G_i$ in $R$ and edges of $G_j$ in $R'$ is bounded by $\mathcal{O}(1/\varepsilon^2)$. Moreover, the number of intersecting edges is zero if $R$ and $R'$ are disjoint. We show that only $\mathcal{O}(N)$ pairs of regions $R \in D_i'$ and $R' \in D_j'$ overlap, so that the total number of intersection points of edges in $G_i$ with edges in $G_j$ is $\mathcal{O}(N/\varepsilon^2)$.

To show that there are only $\mathcal{O}(N)$ pairs of overlapping regions in subdivisions $D_i'$ and $D_j'$, we construct an overlap graph $H$. The vertex set of graph $H$ contains one vertex per region in subdivisions $D_i'$ and $D_j'$. There is an edge from a vertex $v$ to a vertex $w$, if region $R(v)$ overlaps region $R(w)$, and $\ell_{\max}(R(v)) \leq \ell_{\max}(R(w))$, where $R(v)$ and $R(w)$ are the regions corresponding to vertices $v$ and $w$. The number of pairs of overlapping regions is bounded by the number of edges in graph $H$. We show that every vertex in graph $H$ has constant out-degree, so that the edge set of graph $H$ has size $\mathcal{O}(N)$ because subdivisions $D_i'$ and $D_j'$ consist of $\mathcal{O}(N)$ regions each, by Lemma 14.6.

So let $R$ be a region in $D_i'$. We need to bound the number of regions $R'$ in $D_j'$ overlapping $R$ and so that $\ell_{\max}(R') \geq \ell_{\max}(R)$. Consider the coordinate system used to define subdivision $D_j'$. Then region $R$ is contained in an axes-parallel rectangle $\bar{R}$ whose sides have length at most $\sqrt{2}\ell_{\max}(R)$. Every region $R'$ overlapping $R$ must overlap $\bar{R}$. Since the bounding rectangle of every region $R'$ is a box, we have $\ell_{\min}(R') \geq \frac{1}{3}\ell_{\max}(R') \geq \frac{1}{3}\ell_{\max}(R) \geq \frac{1}{3\sqrt{2}}\ell_{\max}(\bar{R})$. Now observe that all regions in $D_j'$ are disjoint

and that every donut cell fills at least one ninth of the area of its bounding box. Hence, the number of regions of $D_j$ overlapping $\bar{R}$ is bounded by $9(3\sqrt{2}+1)^2 = \mathcal{O}(1)$, which finishes the proof. □

We have to show how to construct graph $G$ I/O-efficiently. In order to compute graph $G$, we apply the following procedure, starting with graphs $G_0, \ldots, G_q$, and repeating it until a single graph remains, which is graph $G$: We form pairs of graphs in the current set of graphs. For each pair $(G', G'')$, we apply the red-blue line segment intersection algorithm of [15] to compute the set of intersection points between the edges in $G'$ and $G''$. We apply procedure DUPLICATEREMOVAL to the union of the vertex sets of graphs $G'$ and $G''$ with the set of intersection points to obtain the vertex set of the superimposition $G^\circ$ of graphs $G'$ and $G''$. The red-blue line segment intersection algorithm can be augmented so that it labels every intersection point with the two edges of graphs $G'$ and $G''$ intersecting in this point. Hence, we can apply the procedure from page 302 to obtain the edge set of $G^\circ$. This procedure computes graph $G^\circ$ in $\mathcal{O}\left(\frac{|G^\circ|}{DB}\log_{\frac{M}{DB}}\frac{|G^\circ|}{DB}\right)$ I/Os.

Using the above procedure, we compute a hierarchy of graphs where graphs $G_0, \ldots, G_q$ are at level 0, and every graph at level $i$ is produced by superimposing at most two graphs at level $i-1$. Every graph at level $i$ is the superimposition of at most $2^i$ level-0 graphs. Hence, by the same arguments as in the proof of Lemma 14.18, the size of a level-$i$ graph is at most $\mathcal{O}(2^{2i}N/\varepsilon^2)$. On the other hand, the number of these graphs can be bounded by $\lceil 3\pi/(2^i\varepsilon)\rceil$. Hence, one level of the hierarchy of graphs can be computed in $\mathcal{O}\left(\frac{2^iN/\varepsilon^3}{DB}\log_{\frac{M}{DB}}\frac{2^iN/\varepsilon^3}{DB}\right)$ I/Os, so that the computation of graph $G$ takes $\sum_{i=0}^{h}\mathcal{O}\left(\frac{2^iN/\varepsilon^3}{DB}\log_{\frac{M}{DB}}\frac{2^iN/\varepsilon^3}{DB}\right) = \mathcal{O}\left(\frac{N/\varepsilon^4}{DB}\log_{\frac{M}{DB}}\frac{N/\varepsilon^4}{DB}\right)$ I/Os, where $h = \lceil\log(3\pi/\varepsilon)\rceil$ is the number of levels in the hierarchy. This proves the following result.

**Theorem 14.3** *Given a set $P$ of polygonal obstacles with $N$ vertices, a planar $L_2$-Steiner spanner of size $\mathcal{O}(N/\varepsilon^4)$ and with spanning ratio $1+\varepsilon$ can be computed in $\mathcal{O}\left(\frac{N/\varepsilon^4}{DB}\log_{\frac{M}{DB}}\frac{N/\varepsilon^4}{DB}\right)$ I/Os using $\mathcal{O}\left(\frac{N/\varepsilon^4}{B}\log_{\frac{M}{DB}}\frac{N/\varepsilon^4}{DB}\right)$ blocks of external memory.*

# Chapter 15

# Conclusions and Open Problems

The primary focus of this thesis is on exploiting the topology of outerplanar graphs, planar graphs, and spanner graphs of point sets and sets of polygonal obstacles to solve shortest path problems in these graphs I/O-efficiently. We believe that the work in this thesis is a major step toward understanding the I/O-complexity of shortest path problems on sparse graphs.

Two of the main results of Part I of this thesis show that the single source shortest path problem can be solved I/O-efficiently on outerplanar and planar graphs. In [128], it is shown that this is also true for graphs of bounded treewidth. The reason why these graph classes admit I/O-efficient solutions to this problem is that they have small separators and that a separator decomposition of these graphs can be obtained I/O-efficiently. Indeed, the shortest path algorithms for outerplanar graphs, planar graphs, and graphs of bounded treewidth solve the SSSP problem by applying dynamic programming to a separator decomposition of the graph, even though this decomposition has depth one in the case of planar graphs. For outerplanar and planar graphs, we have demonstrated that their geometric nature provides sufficient information to compute the required separator decomposition I/O-efficiently. For graphs of bounded treewidth, the decomposition is obtained from a tree-decomposition of the graph, which can be computed I/O-efficiently [128]. It is an interesting question whether other properties than having small separators can be exploited to solve the

single source shortest path problem on sparse graphs that do not have small separators.

The major open problem in the context of Part I of the thesis is whether the amount of main memory required by the unweighted separator algorithm for planar graphs can be reduced. This algorithm and the shortest path algorithm of [12] are the only steps of the algorithms for planar graphs presented here which require that $M = \omega(DB)$. A closely related open problem is the semi-external shortest path problem. In particular, we ask the question whether there exists an algorithm which solves the single source shortest path problem on general graphs in $\mathcal{O}(\text{sort}(|E|))$ I/Os under the assumption that the vertex set can be held in internal memory. If this is the case, it seems that a bootstrapping approach can be applied to lower the memory requirements of our separator algorithm and the shortest path algorithm of [12] to $M = \mathcal{O}(DB\,\text{polylog}(DB))$ or even $M = \mathcal{O}(DB)$.

The major open problem in the context of Part II of the thesis is whether there exists a sparse spanner graph of a set of polygonal obstacles which does not require the addition of Steiner points, can be computed I/O-efficiently, and allows spanner paths to be reported I/O-efficiently. In [125], we give a partial answer to this question by showing that the $\theta$-graph of [46] can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. Even though it should be possible to exploit the rather regular structure of this spanner to develop an algorithm for reporting spanner paths I/O-efficiently, this appears to be non-trivial.

A minor open problem which we would like to mention is a possible improvement of the algorithms for computing $K$-nearest neighbors and $K$-closest pairs. Our algorithms are close to optimal, as it can be shown that these problems require $\Omega(\text{sort}(N) + \text{scan}(KN))$ and $\Omega(\text{sort}(N) + \text{scan}(K))$ I/Os, respectively. The reason why our algorithms do not match these lower bounds are the permutation problems involved in constructing candidate sets $X(B)$ and $N'(a)$ in the $K$-nearest neighbor algorithm, and the computation of candidate set $C$ in the $K$-closest pair algorithm. It seems that these permutations are of a special type and that it should be possible to use the split tree to route these permutations in $o(\text{perm}(N))$ I/Os.

# Bibliography


1. J. Abello, A. L. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. In *Proceedings of the 6th European Symposium on Algorithms*, pp. 332–343, 1998.

2. P. K. Agarwal, L. Arge, M. Murali, K. R. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 117–126, 1998.

3. P. K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete & Computational Geometry*, 6:407–422, 1991.

4. A. Aggarwal and R. J. Anderson. A random NC algorithm for depth first search. *Combinatorica*, 8(1):1–12, 1988.

5. A. Aggarwal, R. J. Anderson, and M.-Y. Kao. Parallel depth-first search in general directed graphs. *SIAM Journal on Computing*, 19:397–409, 1990.

6. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, pp. 1116–1127, September 1988.

7. R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–233, 1990.

8. L. Aleksandrov and H. Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM Journal of Discrete Mathematics*, 9:129–150, 1996.

9. L. Aleksandrov, H. Djidjev, H. Guo, and A. Maheshwari. Separators of low cost. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments*, 2002. To appear.

10. N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical Report 71/87, Tel-Aviv University, 1987.

11. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the 4th Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pp. 334–345. Springer-Verlag, 1995.

12. L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP, and multi-way planar separators. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pp. 433–447. Springer-Verlag, 2000.

13. L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures*, volume 2125 of *Lecture Notes in Computer Science*, pp. 471–482. Springer-Verlag, 2001.

14. L. Arge and P. B. Miltersen. On showing lower bounds for external-memory computational geometry problems. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. AMS, 1999.

15. L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proceedings of the 3rd Annual European Symposium on Algorithms*, volume 979 of *Lecture Notes in Computer Science*, pp. 295–310. Springer-Verlag, 1995.

16. S. Arikati, D. Z. Chen, L. P. Chew, G. Das, M. Smid, and C. D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. In *Proceedings of the 4th Annual European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pp. 514–528. Springer-Verlag, 1996.

17. S. Arnborg and A. Proskurowski. Linear time algorithms for NP-hard problems restricted to partial $k$-trees. *Discrete Applied Mathematics*, 23:11–24, 1989.

18. S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. Smid. Euclidean spanners: Short, thin, and lanky. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, pp. 489–498, 1995.

19. S. Arya, D. M. Mount, and M. Smid. Randomized and deterministic algorithms for geometric spanners of small diameter. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pp. 703–712, 1994.

20. S. Arya and M. Smid. Efficient construction of bounded-degree spanners with low weight. *Algorithmica*, 17:33–54, 1997.

21. 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*, pp. 17–30, 1995.

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

23. R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):97–90, 1958.

24. M. Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing*, pp. 80–86, 1983.

25. J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23:214–229, 1980.

26. J. L. Bentley and M. I. Shamos. Divide-and-conquer in multidimensional space. In *Proceedings of the 8th Annual ACM Symposium on the Theory of Computing*, pp. 220–230, 1976.

27. H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pp. 105–118. Springer-Verlag, 1988.

28. H. L. Bodlaender and T. Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM Journal on Computing*, 27:1725–1746, 1998.

29. H. L. Bodlaender, G. Tel, and N. Santoro. Trade-offs in non-reversing diameter. *Nordic Journal of Computing*, 1:111–134, 1994.

30. K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

31. J. Boyer and W. Myrvold. Stop minding your P's and Q's: A simplified $O(n)$ planar embedding algorithm. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 140–146, 1999.

32. A. Broder, R. Kumar, F. Manghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proceedings of the 9th International World-Wide Web Conference*, 2000. `http://www9.org`.

33. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pp. 859–860, 2000.

34. P. B. Callahan. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pp. 332–341, 1993.

35. P. B. Callahan. *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, 1995.

36. P. B. Callahan, M. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proceedings of the 4th Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pp. 381–392. Springer-Verlag, 1995.

37. P. B. Callahan and S. R. Kosaraju. A decomposition of multi-dimensional point sets with applications to $k$-nearest-neighbors and $n$-body potential fields. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pp. 546–556, 1992.

38. P. B. Callahan and S. R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 291–300, 1993.

39. P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and $n$-body potential fields. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 263–272, 1995.

40. P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to $k$-nearest neighbors and $n$-body potential fields. *Journal of the ACM*, 42:67–90, 1995.

41. B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. *International Journal of Computational Geometry & Applications*, pp. 125–144, 1995.

42. P. Chew. There is a planar graph almost as good as the complete graph. In *Proceedings of the 2nd Annual ACM Symposium on Computational Geometry*, pp. 169–177, 1986.

43. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 139–149, January 1995.

44. N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *Journal of Computer and System Sciences*, 30(1):54–76, 1985.

45. K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pp. 226–232, 1983.

46. K. L. Clarkson. Approximation algorithms for shortest path motion planning. In *Proceedings of the 19th ACM Symposium on the Theory of Computing*, pp. 56–65, 1987.

47. E. Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21:331–357, 1996.

48. E. Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest-paths. *Journal of the ACM*, 47:132–166, 2000.

49. R. Cole and M. T. Goodrich. Optimal parallel algorithms for polygon and point-set problems. In *Proceedings of the 4th Annual ACM Symposium on Computational Geometry*, pp. 201–210, 1988.

50. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, first edition, 1990.

51. A. Crauser, F. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. In *Proceedings of the 14th Annual ACM Symposium on Computational Geometry*, pp. 259–268, 1998.

52. A. Crauser, K. Mehlhorn, and U. Meyer. Kürzeste-Wege-Berechnung bei sehr großen Datenmengen. In O. Spaniol, editor, *Promotion tut not: Innovationsmotor "Graduiertenkolleg"*, volume 21 of *Aachener Beiträge zur Informatik*. Verlag der Augustinus Buchhandlung, 1996.

53. G. Das, P. Heffernan, and G. Narasimhan. Optimally sparse spanners in 3-dimensional Euclidean space. In *Proceedings of the 9th Annual ACM Symposium on Computational Geometry*, pp. 53–62, 1993.

54. G. Das and G. Narasimhan. A fast algorithm for constructing sparse Euclidean spanners. *International Journal of Computational Geometry & Applications*, 7:297–315, 1997.

55. G. Das, G. Narasimhan, and J. S. Salowe. A new way to weigh malnourished Euclidean graphs. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 215–222, 1995.

56. F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pp. 106–115, 1997.

57. F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996.

58. M. T. Dickerson, R. L. S. Drysdale, and J.-R. Sack. Simple algorithms for enumerating interpoint distances and finding $k$ nearest neighbors. *International Journal of Computational Geometry & Applications*, 2:221–239, 1993.

59. M. T. Dickerson and D. Eppstein. Algorithms for proximity problems in higher dimensions. *Computational Geometry: Theory and Applications*, 5:277–291, 1996.

60. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.

61. K. Diks, H. N. Djidjev, O. Sykora, and I. Vrto. Edge separators of planar and outerplanar graphs with applications. *Journal of Algorithms*, 14:258–279, 1993.

62. H. Djidjev, G. Pantziou, and C. Zaroliagis. Computing shortest paths and distances in planar graphs. In *Proceecings of the 18th Internatonal Conference on Algorithms, Languages, and Programming*, volume 510 of *Lecture Notes in Computer Science*, pp. 327–339. Springer-Verlag, 1991.

63. H. N. Djidjev. Partitioning graphs with costs and weights on vertices: Algorithms and applications. *Algorithmica*, 28:51–75, 2000.

64. H. N. Djidjev and J. R. Gilbert. Separators in graphs with negative and multiple vertex weights. *Algorithmica*, 23:57–71, 1999.

65. H. N. Djidjev, G. E. Pantziou, and C. D. Zaroliagis. Improved algorithms for dynamic shortest paths. *Algorithmica*, 28:367–389, 2000.

66. D. Dobkin, S. J. Friedman, and K. J. Supowit. Delaunay graphs are almost as good as complete graphs. *Discrete & Computational Geometry*, 5:399–407, 1990.

67. D. Eppstein. Spanning trees and spanners. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. North-Holland, 2000.

68. D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer. Separator based sparsification I. Planarity testing and minimum spanning trees. *Journal of Computer and System Sciences*, 52:3–27, 1996.

69. S. Even. *Graph Algorithms*. Computer Science Press, 1979.

70. S. Even and R. E. Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2:339–344, 1976.

71. R. W. Floyd. Algorithm 97 (SHORTEST PATHS). *Communications of the ACM*, 5(6):345, 1962.

72. L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

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

74. G. N. Frederickson. Planar graph decomposition and all pairs shortest paths. *Journal of the ACM*, 38(1):162–204, January 1991.

75. G. N. Frederickson. Using cellular embeddings in solving all pairs shortest paths problems. *Journal of Algorithms*, 19:45–85, 1995.

76. G. N. Frederickson. Searching among intervals in compact routing tables. *Algorithmica*, 15:448–466, 1996.

77. G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24:37–65, 1997.

78. M. Fredman, F. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31:538–544, 1984.

79. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

80. A. M. Frieze, G. L. Miller, and S.-H. Teng. Separator based parallel divide and conquer in computational geometry. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 420–429, 1992.

81. H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

82. Z. Galil, G. F. Italiano, and N. Sarnak. Fully dynamic planarity testing. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pp. 495–506, 1992.

83. H. Gazit and G. L. Miller. A parallel algorithm for finding a separator in planar graphs. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pp. 238–248, 1987.

84. H. Gazit and G. L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Information Processing Letters*, 28:61–65, 1988.

85. R. K. Ghosh and G. P. Bhattacharjee. Parallel breadth-first search algorithms for trees and graphs. *International Journal of Computer Mathematics*, 15:255–268, 1984.

86. J. R. Gilbert, J. P. Hutchinson, and R. E. Tarjan. A separator theorem for graphs of bounded genus. *Journal of Algorithms*, 5:391–407, 1984.

87. M. Golin, R. Raman, C. Schwarz, and M. Smid. Randomized data structures for the dynamic closest-pair problem. *SIAM Journal on Computing*, 27:1036–1072, 1998.

88. M. T. Goodrich. Planar separators and parallel polygon triangulation. *Journal of Computer and System Sciences*, 51(3):374–389, 1995.

89. M. T. Goodrich, Y. Matias, and U. Vishkin. Optimal parallel approximation algorithms for prefix sums and integer sorting. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 241–250, 1994.

90. M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, November 1993.

91. S. Govindarajan, T. Lukovszki, A. Maheshwari, and N. Zeh. I/O-efficient well-separated pair decomposition and its applications. In *Proceedings of the 8th Annual European Symposium on Algorithms*, volume 1879 of *Lecture Notes in Computer Science*, pp. 220–231. Springer-Verlag, September 2000.

92. T. Hagerup. Planar depth-first search in $O(\log n)$ parallel time. *SIAM Journal on Computing*, 19(4):678–704, August 1990.

93. F. Harary. *Graph Theory.* Addison-Wesley, 1969.

94. X. He and Y. Yesha. A nearly optimal parallel algorithm for constructing depth first spanning trees in planar graphs. *SIAM Journal on Computing*, 17:486–491, 1988.

95. K. Hinrichs, J. Nievergelt, and P. Schorn. Plane-sweep solves the closest pair problem elegantly. *Information Processing Letters*, 26:255–261, 1987/88.

96. K. Hinrichs, J. Nievergelt, and P. Schorn. An all-round sweep algorithm for 2-dimensional nearest-neighbors problems. *Acta Informatica*, 26:383–394, 1992.

97. J. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2:135–158, 1973.

98. J. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

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

100. D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. In *Proceedings of the 5th ACM-SIAM Computing and Combinatorics Conference*, volume 1627 of *Lecture Notes in Computer Science*, pp. 51–60. Springer-Verlag, July 1999. To appear in Discrete Applied Mathematics.

101. J. JáJá and R. Kosaraju. Parallel algorithms for planar graph isomorphism and related problems. *IEEE Transactions on Circuits and Systems*, 35(3):304–311, March 1988.

102. J. JáJá and J. Simon. Parallel algorithms in graph theory: Planarity testing. *SIAM Journal on Computing*, 11(2):313–328, 1982.

103. D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

104. M.-Y. Kao. Planar strong connectivity helps in parallel depth-first search. *SIAM Journal on Computing*, 24:46–62, 1995.

105. M.-Y. Kao, S.-H. Teng, and K. Toyama. An optimal parallel algorithm for planar cycle separators. *Algorithmica*, 14:398–408, 1995.

106. D. Kavvadias, G. E. Pantziou, P. G. Spirakis, and C. D. Zaroliagis. Hammock-on-ears decomposition: A technique for the efficient parallel solution of shortest paths and other problems. *Theoretical Computer Science*, 168(1):121–154, 1996.

107. J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete Euclidean graph. *Discrete & Computational Geometry*, 7:13–28, 1992.

108. S. Khuller and Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118:34–37, 1995.

109. T. Kim and K. Chwa. Parallel algorithms for a depth first search and a breadth first search. *International Journal of Computer Mathematics*, 19:39–52, 1986.

110. P. Klein. On Gazit and Miller's parallel algorithm for planar separators: Achieving greater efficiency through random sampling. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 43–49, 1993.

111. P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55:3–23, 1997.

112. P. Klein and J. Reif. An efficient parallel algorithm for planarity. *Journal of Computer and System Sciences*, 37:190–246, 1988.

113. P. N. Klein and S. Sairam. A parallel randomized approximation scheme for shortest paths. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pp. 750–758, 1992.

114. P. N. Klein and S. Subramanian. A linear-processor polylog-time algorithm for shortest-paths in planar graphs. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pp. 259–270, 1993.

115. M. van Kreveld. Algorithms on triangulated terrains. In *Proceedings of the 24th SOFSEM*, volume 1338 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

116. M. van Kreveld. Digital elevation models and TIN algorithms. In *Algorithmic Foundations of GIS*, volume 1340 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

117. J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

118. V. Kumar and E. 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 Processing*, pp. 169–176, October 1996.

119. M. Lanthier. *Shortest Path Problems on Polyhedral Surfaces.* PhD thesis, School of Computer Science, Carleton University, Ottawa/Canada, December 1999.

120. M. Lanthier, A. Maheshwari, and J.-R. Sack. Approximating weighted shortest paths on polyhedral surfaces. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry*, pp. 274–283, June 1997.

121. M. Lanthier, A. Maheshwari, and J.-R. Sack. Approximating shortest paths on weighted polyhedral surfaces. *Algorithmica*, to appear.

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

123. H.-P. Lenhof and M. Smid. Sequential and parallel algorithms for the $k$ closest pairs problem. *International Journal of Computational Geometry & Applications*, 5:273–288, 1995.

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

125. T. Lukovszki, A. Maheshwari, and N. Zeh. I/O-efficient batched range counting and its applications to proximity problems. In *Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pp. 244–255. Springer-Verlag, 2001.

126. A. Maheshwari, M. Smid, and N. Zeh. I/O-efficient shortest path queries in geometric spanners. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures*, volume 2125 of *Lecture Notes in Computer Science*, pp. 287–299. Springer-Verlag, 2001.

127. A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proceedings of the 10th International Symposium on Algorithms and Computation*, volume 1741 of *Lecture Notes in Computer Science*, pp. 307–316. Springer-Verlag, December 1999.

128. A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 89–90, 2001.

129. A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 372–381, 2002.

130. Y. Matias and U. Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12:573–606, 1991.

131. K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.

132. U. Meyer. External memory BFS on undirected graphs with bounded degree. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 87–88, 2001.

133. G. Miller and J. Reif. Parallel tree contraction and its applications. In *Proceedings of the 26th IEEE Annual Symposium on Foundations of Computer Science*, pp. 478–489, 1985.

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

135. S. L. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Information Processing Letters*, 9(5):229–232, December 1979.

136. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 687–694, January 1999.

137. G. Narasimhan and M. Smid. *Geometric Spanner Networks*. Cambridge University Press, To appear.

138. M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 120–129, June/July 1993.

139. V. Pan and J. H. Reif. Fast and efficient solutions of path algebra problems. *Journal of Computer and System Sciences*, 38:494–510, 1989.

140. G. Pantziou, P. Spirakis, and C. Zaroliagis. Efficient parallel algorithms for shortest paths in planar digraphs. *BIT*, 32:215–232, 1992.

141. F. P. Preparata and M. I. Shamos. *Computational Geometry — An Introduction.* Springer-Verlag, 1985.

142. R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

143. W. Pugh. Skip lists: A probabilistic alternative to balanced search trees. *Communications of the ACM*, 33:668–676, 1990.

144. M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pp. 21–39. Academic Press, 1976.

145. V. Ramachandran and J. H. Reif. Planarity testing in parallel. *Journal of Computer and System Sciences*, 49(3):517–561, 1994.

146. J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, June 1985.

147. C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

148. J. Ruppert and R. Seidel. Approximating the $d$-dimensional complete Euclidean graph. In *Proceedings of the 3rd Canadian Conference on Computational Geometry*, pp. 207–210, 1991.

149. K. Salem and H. Garcia-Molina. Disk striping. In *Proceedings of the 2nd IEEE Data Engineering Conference*, pp. 336–342, 1986.

150. J. S. Salowe. Constructing multidimensional spanner graphs. *International Journal on Computational Geometry & Applications*, 1:99–107, 1991.

151. J. S. Salowe. Enumerating interdistances in space. *International Journal of Computational Geometry & Applications*, 2:49–59, 1992.

152. P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 849–858, 2000.

153. R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, pp. 37–67. Springer-Verlag, 1993.

154. M. I. Shamos. Geometric complexity. In *Proceedings of the 7th Annual ACM Symposium on the Theory of Computing*, pp. 224–233, 1975.

155. M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science*, pp. 151–162, 1975.

156. G. E. Shannon. A linear-processor algorithm for depth-first search in planar graphs. *Information Processing Letters*, 29:119–123, 1988.

157. D. D. Sleator and R. E. Tarjan. A data sturcture for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

158. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

159. M. Smid. Maintaining the minimal distance of a point set in less than linear time. *Algorithms Review*, 2:33–44, 1991.

160. M. Smid. Closest-point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. North-Holland, 2000.

161. M. Smid. Private communication, 2000.

162. J. R. Smith. Parallel algorithms for depth-first searches I. Planar graphs. *SIAM Journal on Computing*, 15(3):814–830, August 1986.

163. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–159, 1972.

164. M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.

165. M. Thorup. Floats, integers, and single source shortest paths. *Journal of Algorithms*, 35:189–201, 2000.

166. J. L. Träff and C. D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. *Journal of Parallel and Distributed Computing*, 60:1103–1124, 2000.

167. P. M. Vaidya. Minimum spanning trees in $k$-dimensional space. *SIAM Journal on Computing*, 17:572–582, 1988.

168. P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete & Computational Geometry*, 4:101–115, 1989.

169. P. M. Vaidya. A sparse graph almost as good as the complete graph on points in $K$ dimensions. *Discrete & Computational Geometry*, 6:369–381, 1991.

170. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

171. J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33, 2001.

172. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

173. S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

174. H. Whitney. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54:150–168, 1932.

175. H. Whitney. Non-separable and planar graphs. *Transaction of the American Mathematical Society*, 34:339–362, 1932.

176. A. C. Yao. On constructing minimum spanning trees in $k$-dimensional space and related problems. *SIAM Journal on Computing*, 11:721–736, 1982.

177. N. Zeh. *An External-Memory Data Structure for Shortest Path Queries.* Diplomarbeit, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, November 1998.

178. N. Zeh. I/O-efficient planar embedding using graph separators. Technical Report TR-01-07, School of Computer Science, Carleton University, Ottawa, Canada, 2001.

179. N. Zeh. I/O-efficient planar separators and applications. Technical Report TR-01-02, School of Computer Science, Carleton University, Ottawa, Canada, 2001.