I/O-Efficient Graph Algorithms

Norbert Zeh

Department of Computer Science, Duke University, Durham, NC 27708-0129, USA
nzeh@cs.duke.edu

1 Introduction

Graph algorithms are fundamental in the sense that many general problems or problems in computational geometry can be reduced to graph problems. For example, a number of parallel scheduling problems can be reduced to the problem of finding a maximum or minimum matching of a bipartite graph. Shortest path problems in geometric domains are often solved by computing shortest paths in graphs that capture the geometric distances between well-chosen discrete locations. One may even go as far as saying that most pointer-based data structures are just graphs with additional information stored at their vertices. Extracting information from such a data structure then becomes a graph problem. For example, a standard search query on a binary search tree T is transformed into the problem of traversing a path in this tree.

Now large data structures used in large scale applications do not provide a fertile ground for interesting graph problems to be studied. So one question to ask is whether there are other real-life applications where massive graphs need to be handled. Two important areas where massive graphs arise are web-modelling and geographic information systems. The graphs produced by recent web crawls have on the order of 200 million vertices and 2 billion edges, which can be processed in main memory only by machines at the higher end of the price scale. But current work in web-modelling studies the structure of the web by performing breadth-first search (BFS) and depth-first search (DFS) in these graphs or computing their connected components. Some of the problems arising in geographic information systems include flow problems on terrains and logistics and road planning applications that involve the computation of shortest paths on weighted terrains. These terrains are often represented by large maps at a fine granularity, which use gigabytes of storage. Many of these applications reduce the given real-life problem to computations in graphs that approximate the given surface sufficiently well. So what is left is a graph problem.

The above is meant to give a motivation for the study of I/O-efficient graph algorithms. In this lecture series we discuss algorithms for fundamental problems such as BFS, DFS, shortest paths and connectivity, with a focus on the techniques that lead to I/O-efficient algorithms for these problems. In the first, and larger, part of the lecture series we concentrate on general approaches that lead to efficient algorithms without additional information about the structure of the given graph. In the second part we study algorithms for planar graphs, which is one of the classes of sparse graphs for which improved algorithms have been developed. In light of the above discussion, this class of graphs is important because the graphs that arise in shortest path computations on weighted terrains are "almost planar", and many ideas used in algorithms for planar graphs can be used to solve the problems discussed here on such "almost planar" graphs.

The following is a list of the problems we consider, sorted by the sections where they are discussed:

Problem	**Section**
List ranking	2
Euler tours and algorithms for trees	3
Evaluating DAGs and greedy algorithms for undirected graphs	4
Graph contraction and applications to connectivity problems (connected components, minimum spanning tree, etc.)	5
Breadth-first search and depth-first search	6, 10
Single source shortest paths	7, 8
Planar graph partitions	9

In Section 12, we sketch the ideas of solutions to a few more problems on sparse graphs and outline some of the most important and most challenging open problems in the area of I/O-efficient graph algorithms.

We assume that the reader is familiar with elementary graph theoretic concepts such as the definitions of directed and undirected graphs, acyclicity of directed graphs, adjacency of vertices, or independent sets and maximal matchings. For good introductory texts on graph theory, the reader may refer to [15, 17, 31].

2 List Ranking

The first problem we discuss is list ranking, which has proved to be an important tool in parallel algorithms. Given the similarity between the problems arising in the design of parallel and I/O-efficient algorithms, an I/O-efficient list ranking algorithm can be used to obtain I/O-efficient algorithms for a wide range of problems on simple graphs such as trees. List ranking is also a nice introductory example to demonstrate how surprisingly difficult even extremely simple graph problems can become, once random memory access is penalized.

The *list ranking* problem is the following: *Given a linked list L, compute for every element of L its distance from the head of L.*[1] To cast this problem in

[1] Originally, the rank of an element was defined as its distance from the tail of the list. However, it is an exercise to verify that an algorithm that can compute either of the two distances can compute the other. The definition used here simplifies the discussion.

Procedure NAÏVELISTRANKING

```
1: $v \leftarrow h$
2: $\rho \leftarrow 0_l$                              {$0_l$ is the left-neutral element w.r.t. $\oplus$.}
3: while $v \neq$ nil do
4:     $\rho \leftarrow \rho \oplus \omega(v)$
5:     $\rho(v) \leftarrow \rho$
6:     $v \leftarrow \mathrm{succ}(v)$
7: end while
```

Algorithm 2.1
A linear-time internal memory list ranking algorithm.

graph theoretic terms, list L is a directed acyclic graph $L = (V, E)$ with vertex set $V = \{v_1, \ldots, v_N\}$. There are two distinguished vertices h and t, which we call the *head* and *tail* of L. Every vertex except h has exactly one in-edge. Every vertex except t has exactly one out-edge. We assume in this section that the edge set of L is represented implicitly. That is, every vertex $v \in L$ stores a pointer $\mathrm{succ}_L(v) = w$, where $(v, w) \in L$. We call vertex w the *successor* of v in L. For the tail t of L, $\mathrm{succ}_L(t) = \mathbf{nil}$, which signifies that t has no successor in L. If list L is clear from the context, we write $\mathrm{succ}(v)$ instead of $\mathrm{succ}_L(v)$ to denote the successor of v in L. Now let $\sigma : [1, N] \to [1, N]$ be a permutation so that for $1 \leq i < N$, $\mathrm{succ}\left(v_{\sigma(i)}\right) = v_{\sigma(i+1)}$. Then the *rank* of vertex $v_{\sigma(i)}$ is defined as $\rho\left(v_{\sigma(i)}\right) = i$.

As an algorithmic tool, it is often useful to generalize the list ranking problem by adopting the notion of *weighted* ranks of the elements in L: Assume that $\omega : V \to X$ is an assignment of weights drawn from a domain X to the vertices of L, and let $\oplus : X \times X \to X$ be an associative operator on X. Then the *weighted rank* $\rho(v_i)$ of vertex v_i is defined as follows: $\rho\left(v_{\sigma(1)}\right) = \omega\left(v_{\sigma(1)}\right)$. For $1 < i \leq N$, $\rho\left(v_{\sigma(i)}\right) = \rho\left(v_{\sigma(i-1)}\right) \oplus \omega\left(v_{\sigma(i)}\right)$.

Since this somewhat formal definition includes a permutation σ of the vertices of list L, it seems that the list ranking problem contains some formulation of a permutation problem as a subproblem. Hence, we should not be too surprised that list ranking requires $\Omega(\mathrm{perm}(N))$ I/Os. We will see later in this lecture series how to prove this lower bound. But first we investigate why the naïve internal memory algorithm (Algorithm 2.1) is not I/O-efficient. The algorithm makes the assumption that a pointer to the head h of list L is provided.

To see why procedure NAÏVELISTRANKING is not I/O-efficient, assume that $B = 2$ and $M = 4$, and consider the layout of list L shown in Figure 2.1. In order to access the head of list L, the first block has to be loaded into internal memory. The second vertex can only be accessed after loading the second block. In order to access the third vertex, the third block has to be loaded into internal memory. Since there is room for only two blocks in internal memory, one of the blocks already in main memory needs to be discarded. If the LRU[2] paging strategy

[2] LRU stands for "least recently used". That is, to make room for a new block to be loaded into main memory, the block to be discarded from main memory is chosen

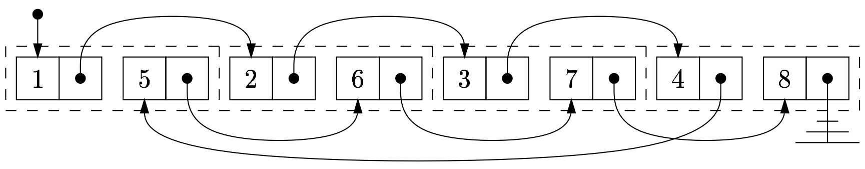

Figure 2.1
A worst-case layout of a list L for procedure NaïveListRanking with LRU paging strategy.

is used, the first block is dropped. Following the execution of the algorithm further, it is not hard to see that the algorithm has to spend one I/O per vertex because just before visiting the vertex, the block containing it is not in internal memory. This example can easily be generalized to blocks of arbitrary size, so that procedure NaïveListRanking spends $\Omega(N)$ I/Os in the worst case.

Chiang *et al.* [12] propose an I/O-efficient list ranking algorithm based on graph contraction (Algorithm 2.2). If the list fits into internal memory, the algorithm loads the whole list into memory and ranks it using procedure NaïveListRanking. Otherwise the algorithm constructs a list L' of size at most $\frac{2}{3}|L|$ by removing the elements of a large independent set I from L. The weights of all elements in $L \setminus I$ are updated so that their weighted ranks in L and L' are the same. Hence, the recursive application of procedure FastListRanking to list L' assigns the correct ranks to all elements in $L \setminus I$. In order to compute the ranks of all elements in I, their weights are added to the ranks of their predecessors.

If $|L| \leq M$, the algorithm spends $\mathcal{O}(\text{scan}(|L|))$ I/Os to rank list L. In particular, list L is read into internal memory in $\mathcal{O}(\text{scan}(|L|))$ I/Os, procedure NaïveListRanking is applied in internal memory, and the ranks of the elements of L are written to disk in $\mathcal{O}(\text{scan}(|L|))$ I/Os. If $|L| > M$, we show below that, excluding the recursive invocation of the algorithm in Line 20, procedure FastListRanking takes $\mathcal{O}(\text{sort}(|L|))$ I/Os. List L', which is passed to the recursive invocation of the algorithm, has size at most $\frac{2}{3}|L|$, so that we obtain the following recurrence describing the I/O-complexity of procedure FastListRanking:

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

The solution of this recurrence is $\mathcal{I}(N) = \mathcal{O}(\text{sort}(N))$, so that procedure FastListRanking is optimal, given the $\Omega(\text{perm}(N))$ I/O lower bound for this problem discussed later in this course.[3]

by the time that has elapsed since the last access to the block. The block with the longest elapsed time since the last access is discarded.

[3] Technically, there is a gap between the upper and lower bounds in the case when $N < \text{sort}(N)$, which is true only for ridiculously large inputs. To satisfy the theo-

Procedure FastListRanking

```
1: if $|L| \le M$ then
2:   Load list $L$ into main memory, and use procedure NaïveListRanking to compute the ranks of all elements in $L$.
3: else
4:   Find an independent set $I$ of size at least $N/3$ in $L$.
5:   for all $v \in L \setminus I$ do
6:     $\mathrm{succ}_{L'}(v) \leftarrow \mathrm{succ}_L(v)$
7:     $\rho_{L'}(v) \leftarrow \rho_L(v)$
8:   end for
9:   for all $v \in I$ do
10:    if $\mathrm{succ}_L(v) \neq$ nil then
11:      $\omega_{L'}(\mathrm{succ}_L(v)) \leftarrow \omega_L(v) \oplus \omega_L(\mathrm{succ}_L(v))$
12:    end if
13:  end for
14:  for all $v \notin I$ do
15:    if $\mathrm{succ}_L(v) \neq$ nil and $\mathrm{succ}_L(v) \in I$ then
16:      $\mathrm{succ}_{L'}(v) \leftarrow \mathrm{succ}_L(\mathrm{succ}_L(v))$
17:    end if
18:  end for
19:  Let $L'$ be the list defined by the vertices in $L \setminus I$, pointers $\mathrm{succ}_{L'}(v)$ and weights $\omega_{L'}(v)$.
20:  Recursively apply procedure FastListRanking to list $L'$. Let $\rho_{L'}(v)$ be the rank assigned to every element $v$ in $L \setminus I$.
21:  for all $v \notin I$ do
22:    $\rho_L(v) \leftarrow \rho_{L'}(v)$
23:    if $\mathrm{succ}_L(v) \neq$ nil and $\mathrm{succ}_L(v) \in I$ then
24:      $\rho_L(\mathrm{succ}_L(v)) \leftarrow \rho_L(v) \oplus \omega_L(\mathrm{succ}_L(v))$
25:    end if
26:  end for
27: end if
```

Algorithm 2.2
An I/O-efficient list ranking procedure.

So let us analyze the I/O-complexity of Lines 4–26 of the algorithm, excluding the recursive call to the algorithm itself in Line 20. We show in Section 4.2 that an independent set of size at least $\frac{2}{3}|L|$ can be found in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. Lines 5–8 can be carried out in a single scan of list L. To carry out Lines 9–13, assume that every vertex has a unique numerical ID. Sort the elements in $L \setminus I$ by their numbers and the elements in I by the numbers of their successors and scan both lists simultaneously to add for every element of I, its weight to the weight of

retician, the gap can be closed by simulating procedures NaïveListRanking and FastListRanking in parallel, allowing each algorithm to perform one I/O before switching to the other algorithm. By stopping the simulation as soon as one of the two algorithms is finished it is guaranteed that the simulation finishes after $\mathcal{O}(\min(N, \mathrm{sort}(N))) = \mathcal{O}(\mathrm{perm}(N))$ I/Os.

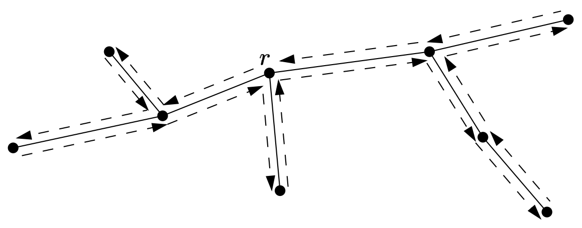

Figure 3.1
An Euler tour of a tree T.

its successor. To carry out Lines 14–18, sort the elements in I by their numbers and the elements in $L \setminus I$ by the numbers of their successors and scan the two sorted lists to update the successors of all elements in $L \setminus I$. Lines 21–26 can be carried out in a similar fashion as Lines 14–18. Except for computing the independent set I, which takes $\mathcal{O}(\text{sort}(N))$ I/Os, this procedure sorts and scans lists of size $\mathcal{O}(N)$ a constant number of times. Hence, the I/O-complexity of one recursive step of procedure FastListRanking is $\mathcal{O}(\text{sort}(N))$ as claimed. Using the above recurrence, this proves the following result.

Theorem 2.1. *A list of size N can be ranked in $\mathcal{O}(\text{sort}(N))$ I/Os.*

Remark. Note that procedure FastListRanking does not make use of the fact that there is a unique head and a unique tail in list L. This allows the algorithm to be applied simultaneously to a collection of linked lists. This fact is exploited by a number of algorithms that use list ranking as a primitive to solve more complicated graph problems.

3 Algorithms for Trees

3.1 The Euler Tour Technique

Before moving on to more complex graph problems, we discuss a simple technique that turns the list ranking algorithm of Section 2 into a powerful tool for solving problems on trees. The goal of this technique is to represent a tree T as a list L so that a number of labelling problems can be solved on T by computing the weighted ranks of the elements in L.

Given a tree T and a distinguished vertex r of T, an *Euler tour* of T is defined as a traversal of T that starts and ends at r and traverses every edge exactly twice, once in each direction (see Figure 3.1). Formally, every undirected edge $\{v, w\} \in T$ is replaced with two directed edges (v, w) and (w, v). The tour starts with an edge (r, v). For every vertex $v \in T$ with incoming edges $e_1, \dots, e_k$ and outgoing edges $e'_1, \dots, e'_k$, numbered so that for $1 \le i \le k$, e_i and e'_i have the same endpoints, edge e_i is succeeded by edge $e'_{(i \bmod k)+1}$ in the tour. When

```
Procedure RootTree
1: Compute an Euler tour $L$ of tree $T$.
2: Compute the rank of every edge $e$ in $L$.
3: for every edge $\{v, w\} \in T$ do
4:     Store the ranks of edges $(v, w)$ and $(w, v)$ in $L$ with edge $\{v, w\}$.
5: end for
```

Algorithm 3.1
Rooting a tree T.

referring to the construction of an Euler tour, we mean the construction of a circular linked list L so that every edge has its successor in the tour as its successor in L.

Before studying the power of this technique for computing labellings of trees, we make the following observation.

Lemma 3.1. *Given a tree T in adjacency list representation, an Euler tour of T can be computed in $\mathcal{O}(\text{scan}(N))$ I/Os. If the edge set of T is represented as an unordered collection of edges, the tour can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

Proof. Given the adjacency list $A(v)$ of a vertex v, the successors in L of all incoming edges of v can be computed in a single scan of list $A(v)$. If the edges of T are given as an unordered set of edges, an adjacency list representation of T can be obtained as follows: First replace every edge $\{v, w\}$ of T by two directed edges (v, w) and (w, v). Then sort the resulting set of directed edges lexicographically. □

3.2 Rooting a Tree

A tree as a data structure is often rooted. That is, it has a distinguished *root vertex* r and a well-defined parent-child relation between adjacent vertices. Trees, when considered as undirected graphs, do not have this structure imposed on them. However, most structural information about a tree T, provided by labelling the vertices of T in a meaningful manner, can be obtained only after declaring one of the vertices to be the root and establishing parent-child and ancestor-descendant relations defined as follows: Let r be the chosen root of T. Then a vertex v is an *ancestor* of a vertex w, and w is a *descendant* of v if v is on the unique path from r to w. If $v \neq w$, v is a *proper* ancestor of w. Vertex v is the *parent* of w, and w is the *child* of v, if v is an ancestor of w and $\{v, w\} \in T$.

Rooting a tree T is the process of choosing a vertex r and labelling the vertices or edges of T so that the labels assigned to two adjacent vertices v and w, or to edge $\{v, w\}$, are sufficient to decide whether v is the parent of w or vice versa. Algorithm 3.1 computes such an edge labelling. In particular, for an edge $\{v, w\} \in T$, v is the parent of w if and only if edge (v, w) has a smaller rank than edge (w, v) in the Euler tour because for every vertex $x \in T$, $x \neq r$, an Euler tour starting at the chosen root r has to traverse edge $(p(x), x)$ before edge $(x, p(x))$.

```
Procedure LabelTree
1: Compute an Euler tour $L$ of $T$ that starts at the root of $T$.
2: Assign appropriate weights to the edges in the Euler tour.
3: Compute the weighted rank of each edge in $L$.
4: Extract a labelling of the vertices of $T$ from these ranks.
```

Algorithm 3.2
Labelling rooted trees.

Theorem 3.2. *A tree T can be rooted in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.*

Proof. We have to show that Algorithm 3.1 takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os. By Theorem 2.1 and Lemma 3.1, Lines 1 and 2 of the algorithm take $\mathcal{O}(\mathrm{sort}(N))$ I/Os. To carry out Lines 3–5, sort the edges in L by their smaller endpoints as a primary key and their larger endpoints as a secondary key. This stores edges (v, w) and (w, v) consecutively, for every edge $\{v, w\} \in T$. Now scan this edge list, replace every pair of edges (v, w) and (w, v) with the corresponding undirected edge $\{v, w\}$, and label edge $\{v, w\}$ with the ranks of both directed edges. □

Remark. A *vertex* labelling that can be used to decide which of two adjacent vertices is the parent can be obtained by assigning to every vertex v the rank of the first edge in L whose source is v. For two adjacent vertices v and w, v is the parent of w if and only if the rank of the first edge with source v is less than the rank of the first edge with source w. This is true because for any vertex $x \in T$, the first edge with source x can be traversed only after visiting vertex x, which in turn is possible only after traversing edge $(p(x), x)$.

3.3 Labelling Rooted Trees

In this section we consider a number of labellings of a rooted tree that provide useful information about the structure of the tree and can be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os using the Euler tour technique and list ranking. Some of these labellings are defined in terms of an Euler tour of the tree that starts at the root. Hence, it is only natural that these labellings can be computed using the Euler tour technique.

Given an Euler tour of a rooted tree T, a *preorder numbering* of T is a numbering of the vertices of T from 1 through N so that a vertex v has a smaller number than another vertex w if and only if the first visit of the tour to vertex v occurs before the first visit to w. A *postorder numbering* of T assigns the smaller number to v if the *last* visit to v occurs before the *last* visit to w. Another important labelling assigns the number of its descendants to every node $v \in T$. Finally, it is handy in a number of applications to know for every vertex $v \in T$, how far away it is from the root. That is, vertex v is to be labelled with the number of edges on the path from r to v in T. We refer to this number as the *depth* of v in T.

Algorithm 3.2 provides a generic method for computing these labellings. The algorithm computes different labellings depending on the choice of the weights assigned to the edges in the Euler tour in Line 2 of the algorithm.

To compute the depth of every vertex v in T, choose the weight $\omega(e)$ of an edge $e = (v, w)$ in L as

$$\omega(e) = \begin{cases} 1 & \text{if } v = p(w) \\ -1 & \text{if } w = p(v) \end{cases}.$$

It is easy to verify that the depth of a vertex v in T equals the weighted rank of any edge (u, v) in the Euler tour. To compute a preorder numbering, choose

$$\omega(e) = \begin{cases} 1 & \text{if } v = p(w) \\ 0 & \text{if } w = p(v) \end{cases}$$

and extract the preorder number of each vertex $v \neq r$ as the rank of edge $(p(v), v)$ plus one. The root r of T always has preorder number 1. A postorder numbering can be computed in a similar fashion. In order to compute the number $|T(v)|$ of descendants of each vertex v, choose the weights of the edges in the Euler tour as for the computation of a preorder numbering, but extract the vertex labels differently. In particular, $|T(r)| = |T| = N$ for the root r of T. For every non-root vertex v, let $r_1(v)$ and $r_2(v)$ be the ranks of edges $(p(v), v)$ and $(v, p(v))$. Then $|T(v)| = r_2(v) - r_1(v) + 1$. From this discussion we obtain the following result.

Theorem 3.3. *The following labellings can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os for a rooted tree with N vertices: a preorder or postorder numbering, a labelling of each vertex with its distance from the root, and a labelling of every vertex with the number of its descendants.*

4 Evaluating Directed Acyclic Graphs

In Sections 2 and 3 we have discussed the most important tools for dealing with lists and trees in an I/O-efficient manner. In this section we turn to a slightly more complicated class of graphs, which can be considered a generalization of lists: directed acyclic graphs (DAGs). The problem we study is that of evaluating a DAG G. More precisely, we are interested in solving the following problem: *Given an assignment of labels $\omega(v)$ to the vertices of G, compute labels $\rho(v)$ of these vertices where the computation of $\rho(v)$ depends only on $\omega(v)$ and labels $\rho(u_1), \dots, \rho(u_k)$ computed for the in-neighbors $u_1, \dots, u_k$ of v.*

Before studying the technique for solving this problem, let us have a look at the list ranking problem again. List ranking is a special case of this evaluation problem where the structure of the DAG and the function that defines labelling ρ are restricted. These restrictions allow an efficient solution of the list ranking problem without making any assumptions about the way the input is

Procedure TimeForwardProcessing

```
1: $Q \leftarrow \emptyset$                                    {$Q$ is a priority queue.}
2: for every vertex $v \in G$, in topologically sorted order do
3:     Let $u_1, \dots, u_k$ be the in-neighbors of $v$.
4:     Retrieve $\rho(u_1), \dots, \rho(u_k)$ from $Q$, using $k$ DeleteMin operations.
5:     Compute $\rho(v)$ from $\omega(v)$ and $\rho(u_1), \dots, \rho(u_k)$.
6:     Let $w_1, \dots, w_l$ be the out-neighbors of $v$.
7:     Insert $l$ copies of $\rho(v)$ into priority queue $Q$. Give the $i$-th copy priority $w_i$.
8: end for
```

Algorithm 4.1
The time-forward processing procedure.

represented. In order to evaluate an *arbitrary* DAG I/O-efficiently, we do have to make a few assumptions. Fortunately these assumptions are satisfied in many interesting applications.

The first assumption we make is that the vertices of G are stored in topologically sorted order. That is, for every edge $(v, w) \in G$, vertex v precedes vertex w in this order. This is crucial because the procedure for evaluating G visits the vertices of G in this order and there is no I/O-efficient algorithm for topologically sorting arbitrary DAGs. That is, if the vertices were arranged in an arbitrary order, the algorithm could end up spending one I/O per vertex when evaluating G or $\Omega(|V|)$ I/Os to topologically sort G, both of which imply that evaluating G would require $\Omega(|V|)$ I/Os.

If there is no bound on the number of in-edges a vertex can have, it is further required that the computation of $\rho(v)$ from $\omega(v)$ and $\rho(u_1), \dots, \rho(u_k)$ can be carried out in $\mathcal{O}(\text{sort}(k))$ I/Os because the evaluation procedure discussed below takes care of providing vertex v with labels $\rho(u_1), \dots, \rho(u_k)$, but cannot carry out the actual computation.

4.1 Time-Forward Processing

Assuming that both assumptions are satisfied, we can now turn to the discussion of a technique for evaluating DAGs. This technique is called *time-forward processing* and was first proposed in [12]. Here we discuss a variant of this technique proposed by Arge [2], which removes a few restrictions of the algorithm of [12] and is surprisingly simple. Algorithm 4.1 shows the pseudo-code. The procedure makes use of a priority queue Q to provide every vertex v with the input required for computing $\rho(v)$. In particular, when a vertex v is evaluated, values $\rho(u_1), \dots, \rho(u_k)$ are retrieved from Q, and $\rho(v)$ is computed from $\omega(v)$ and the retrieved values, either in internal memory or using the $\mathcal{O}(\text{sort}(k))$ I/O algorithm that exists by our second assumption. Once label $\rho(v)$ has been computed, it is inserted into priority queue Q, once for each of the out-neighbors $w_1, \dots, w_k$ of v. The copy of $\rho(v)$ meant for neighbor w_i is inserted with priority w_i.

The correctness and efficiency of this technique now follows from two observations: (1) Every in-neighbor u_i of v is evaluated before v, so that all labels

$\rho(u_1), \ldots, \rho(u_k)$ are inserted into the priority queue before v is evaluated. (2) All vertices preceding v in the topological order are evaluated before v. Thus, their inputs are retrieved from Q before v is evaluated, and the inputs for vertex v are those with smallest priority in Q at the time when v is evaluated. Hence, they can be retrieved using k DELETEMIN operations.

Now it remains to be observed that procedure TIMEFORWARDPROCESSING performs $\mathcal{O}(|E|)$ priority queue operations, one INSERT and one DELETEMIN operation per edge. This implies that the computation of labels $\rho(v)$ from labels $\omega(v)$ takes $\mathcal{O}(\text{sort}(|E|))$ I/Os using an I/O-optimal priority queue [2, 10].

Theorem 4.1. *A DAG $G = (V, E)$ can be evaluated in $\mathcal{O}(\text{sort}(|E|))$ I/Os, provided that the vertices of G are stored in topologically sorted order.*

The fact that the vertices of G have to be given in topologically sorted order is certainly a serious restriction that affects the general applicability of this technique. However, there are many interesting problems on undirected graphs that can be expressed as evaluation problems of appropriate DAGs. In these applications, a topological ordering of the DAG is often easy to obtain by sorting the vertices of the DAG in a natural order induced by the construction of the DAG from the given undirected graph. In the next section we study one such application of the time-forward processing technique to solve a classical problem on undirected graphs.

Remark. Zeh [32] observed that the I/O-complexity of time-forward processing can be reduced to $\mathcal{O}(\text{scan}(|E|))$ if G is a tree and its vertices are stored in preorder or postorder. The idea is to use a stack instead of a priority queue to simulate the sending of information along the edges of G.

4.2 Maximal Independent Set

The problem of computing a maximal independent set of an undirected graph is one representative of a number of problems that can be solved by greedy algorithms of a sufficiently simple structure that they can be simulated using the time-forward processing technique [32]. Recall that an *independent set* of a graph $G = (V, E)$ is a set $I \subseteq V$ of vertices so that no two vertices in I are adjacent. Set I is *maximal* if there is no vertex in $V \setminus I$ that does not have at least one neighbor in I. Procedure MAXIMALINDEPENDENTSET shown in Algorithm 4.2 computes such a set I I/O-efficiently, assuming that every vertex has a unique numerical ID: Lines 2 and 3 of the algorithm clearly take $\mathcal{O}(\text{sort}(|V|+|E|))$ I/Os. The computation of Lines 4–8 can be simulated using time-forward processing: After deciding whether or not a vertex v should be added to set I, vertex v sends a flag to each of its out-neighbors to inform them whether or not v is in I. This way every vertex can decide whether it should be added to I based only on the flags it receives from its in-neighbors. Hence, the whole algorithm takes $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os.

Theorem 4.2. *A maximal independent set of a graph $G = (V, E)$ can be computed in $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os.*

```
Procedure MaximalIndependentSet
1: $I \leftarrow \emptyset$
2: Direct the edges of $G$ from vertices with lower numbers to vertices with higher
   numbers.
3: Sort the vertices of $G$ by their numbers and the edges by the numbers of their
   sources.
4: for every vertex $v \in G$, in sorted order do
5:   if no in-neighbor of $v$ is in $I$ then
6:     Add $v$ to $I$.
7:   end if
8: end for
```

Algorithm 4.2
Computing a maximal independent set of a graph.

Proof. We have already argued that the I/O-complexity of procedure MaximalIndependentSet is $\mathcal{O}(\text{sort}(|V| + |E|))$. The correctness of this procedure follows from the following two observations: Set I as computed by the algorithm is independent because a vertex v is added to I only if none of its in-neighbors is in I. At this point none of its out-neighbors can be in I, and the insertion of v into I prevents all of these out-neighbors from being added to I. Set I is maximal because otherwise there would be a vertex $v \notin I$ none of whose in-neighbors is in I, which implies that v would have been added to I. □

Using Theorem 4.2, we can now fill in the last missing detail of the list ranking algorithm of Section 2. In the description of the algorithm we assumed that an independent set of size at least $N/3$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os for a list of size N. This is shown by the following corollary of Theorem 4.2.

Corollary 4.3. *For a list L of size N, an independent set of size at least $N/3$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

Proof. By Theorem 4.2, a maximal independent set of L can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. However, every maximal independent set of list L has size at least $N/3$ because the vertices in L have degree at most two. □

Two more problems that can be solved using algorithms similar to Algorithm 4.2 are those of computing a maximal matching of a graph G and coloring a graph of degree Δ with at most $\Delta + 1$ colors. The latter problem can be solved using procedure MaximalIndependentSet, only sending different information along the edges of G. The former can be expressed as a vertex-labelling problem of an auxiliary graph, so that an algorithm similar to Algorithm 4.2 can be applied to compute the desired labelling. For details the reader may refer to [32]. It is interesting to observe that a maximal matching corresponds to a maximal independent set of the edge-incidence graph G' of G. The vertices in G' correspond to the edges of G. Two vertices in G' are adjacent if and only if the two corresponding edges in G share an endpoint. Unfortunately graph G' may have size $\Omega(N^2)$ even if G is a tree. Thus, this reduction of the maximal matching problem to that of computing a maximal independent set does not lead to an I/O-efficient maximal matching algorithm.

5 Connectivity Problems

In the rest of this class we study fundamental problems on undirected graphs. We begin in this section with a discussion of connectivity problems such as computing the connected and biconnected components or a minimum spanning tree of a graph. The algorithms for these problems demonstrate the power of an important technique that is applied in a number of I/O-efficient graph algorithms: graph contraction. We discuss this technique in Section 5.1 and turn to concrete applications in Sections 5.2 through 5.4. In Section 5.5 we discuss a special class of graphs for which graph contraction often leads to *I/O-optimal* algorithms.

5.1 The Graph Contraction Paradigm

Graph contraction is a useful technique that was first applied in parallel algorithms. The idea of this technique is simple: Given a graph G and a problem $\mathcal{P}$ to be solved on G, identify (edge-)disjoint subgraphs of G and replace each of them with a smaller subgraph so that a solution of problem $\mathcal{P}$ on G can be derived from a solution of $\mathcal{P}$ on the resulting graph G'. Recursively solve $\mathcal{P}$ on G' and then compute a solution of $\mathcal{P}$ on G from the computed solution on G'.

Of course the recursion cannot continue indefinitely. That is, at some point the algorithm has to stop calling itself recursively and solve problem $\mathcal{P}$ directly. Thus, a contraction-based algorithm $\mathcal{A}$ can be divided into two parts: (1) an algorithm $\mathcal{A}_1$ that constructs graph G' from graph G, calls algorithm $\mathcal{A}$ recursively to solve problem $\mathcal{P}$ on graph G', and then computes a solution of $\mathcal{P}$ on G from the computed solution on G'; (2) an algorithm $\mathcal{A}_2$ that solves problem $\mathcal{P}$ without calling algorithm $\mathcal{A}$. Algorithm $\mathcal{A}$ itself is merely a wrapper that decides which of the two algorithms, $\mathcal{A}_1$ or $\mathcal{A}_2$, to apply to the current input graph. For large inputs, it calls algorithm $\mathcal{A}_1$. For small inputs, it invokes algorithm $\mathcal{A}_2$, thereby stopping the recursion.

The efficiency of the algorithm depends on a number of factors. Clearly the I/O-complexities of algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ have a strong influence on the I/O-complexity of algorithm $\mathcal{A}$. The second important question is how many levels of recursion are needed before algorithm $\mathcal{A}$ stops the recursion by calling algorithm $\mathcal{A}_2$ instead of algorithm $\mathcal{A}_1$. The answer to this question is determined by (1) the ratio between the sizes of graphs G' and G and (2) the largest possible size of graph G so that applying algorithm $\mathcal{A}_2$ to graph G is more efficient than adding another level of recursion by calling algorithm $\mathcal{A}_1$ again. If graph G' has only a constant fraction of the vertices or edges of G, a logarithmic number of recursive calls are sufficient to reduce the size of the graph to a constant, so that algorithm $\mathcal{A}_2$ can solve problem $\mathcal{P}$ in $\mathcal{O}(1)$ I/Os at that point. If algorithm $\mathcal{A}_2$ is more efficient than algorithm $\mathcal{A}_1$ for graphs of more than constant size, the recursion can stop much earlier.

Procedure SemiExternalConnectivity

```
1: Load all vertices of $G$ into main memory and mark each of them as being in its
   own connected component (i.e., $\gamma_G(v) \leftarrow v$, where $\gamma_G(v)$ is a label that identifies
   the connected component of $G$ that contains $v$).
2: for every edge $e \in E$ do
3:   if the two endpoints $v$ and $w$ of $e$ are in different connected components then
4:     Let $\gamma(v)$ and $\gamma(w)$ be the component labels of $v$ and $w$.
5:     for every $u \in V$ do
6:       if $\gamma(u) = \gamma(v)$ or $\gamma(u) = \gamma(w)$ then
7:         $\gamma(u) \leftarrow \min(\gamma(v), \gamma(w))$
8:       end if
9:     end for
10:  end if
11: end for
```

Algorithm 5.1
A semi-external algorithm for connectivity.

5.2 Connectivity

In this section we discuss three different algorithms for computing the connected components of a graph G. Recall that a graph is *connected* if for any two vertices $v, w \in G$, there is a path from v to w in G. The *connected components* of a graph G are its maximal connected subgraphs. The algorithms in this section compute a labelling of the vertices of G so that two vertices have the same label if and only if they belong to the same connected component of G.

The first algorithm we discuss computes the connected components of G I/O-efficiently under the assumption that the vertices, but not the edges, of G fit into main memory. Such an algorithm is often referred to as a *semi-external* algorithm as opposed to a *fully external* algorithm, which assumes that neither the vertex nor the edge set of G fits into main memory.

The second algorithm we discuss is fully external and uses graph contraction to reduce the size of the vertex set of the graph by a factor of two from one level of recursion to the next. As soon as $|V| \leq M$, it calls the semi-external connectivity algorithm to compute the connected components of G without recursing any further.

Finally, the third algorithm is a variation on the second algorithm where the semi-external connectivity algorithm is replaced with an I/O-efficient BFS-algorithm. This allows the recursion to stop much earlier and therefore leads to a more efficient algorithm.

A semi-external connectivity algorithm. If $|V| \leq M$, the connected components of graph G can be computed in $\mathcal{O}(\text{scan}(|V| + |E|))$ I/Os using Algorithm 5.1. The correctness of this algorithm is obvious. To see that the algorithm takes a linear number of I/Os, observe that the vertices of G can be loaded into main memory in $\mathcal{O}(\text{scan}(|V|))$ I/Os. After that, the outer loop requires a scan

```
Procedure ExternalConnectivity
1: if $|V| \le M$ then
2:     Apply procedure SemiExternalConnectivity to compute the connected components of $G$.
3: else
4:     Compute the smallest neighbor $w_v$ for every vertex $v \in G$.
5:     Compute the connected components of the subgraph $H$ of $G$ induced by edges $\{v, w_v\}$, $v \in V$.
6:     Compress each of these connected components into a single vertex. Remove all isolated vertices. Let $G'$ be the resulting graph.
7:     Recursively compute the connected components of $G'$ using procedure ExternalConnectivity.
8:     Re-integrate the isolated vertices into $G'$ and assign a unique label to each such vertex.
9:     For every vertex $v' \in G'$ and every vertex $v$ in the connected component of $H$ represented by $v'$, let $\gamma_G(v) = \gamma_{G'}(v')$.
10: end if
```

Algorithm 5.2
A fully external algorithm for graph connectivity.

of the edge set of G (i.e., $\mathcal{O}(\text{scan}(|E|))$ I/Os), and the inner loop is performed in main memory without incurring any I/Os. Note that the algorithm as presented here is inefficient in terms of the computation it performs in internal memory; but it can easily be made efficient by representing the connected components of G using a union-find data structure [13, Chapter 22] and labelling the vertices only after all connected components have been identified.

A fully external connectivity algorithm. The first fully external connectivity algorithm was proposed by Chiang *et al.* [12] and is shown in Algorithm 5.2. If $|V| \le M$, the algorithm delegates the problem of computing the connected components of G to procedure SemiExternalConnectivity. Otherwise it applies graph contraction to produce a graph G' with at most half as many vertices as G, recursively computes the connected components of G', and derives a labelling of the vertices of G that identifies the connected components of G from the computed labelling of the vertices of G'. Before analyzing the I/O-complexity of procedure ExternalConnectivity, we show that it is correct.

Lemma 5.1. *Let $\gamma_G : V \to \mathbb{N}$ be the component labelling computed by procedure* ExternalConnectivity. *Then for any two vertices $v, w \in G$, $\gamma_G(v) = \gamma_G(w)$ if and only if v and w are in the same connected component of G.*

Proof. We prove the lemma by induction on $|V|$. If $|V| \le M$, the correctness of procedure ExternalConnectivity follows from the correctness of procedure SemiExternalConnectivity. So assume that $|V| = k > M$ and that the algorithm is correct for $|V| < k$. Let $C_1, \dots, C_q$ be the connected components of H, and let G'' be the graph obtained from G by contracting each component C_i

into a single vertex v_i. That is, graph G' is obtained from G'' by removing all isolated vertices. Since $|V(G')| < |V|$, the recursive invocation of procedure ExternalConnectivity on graph G' produces a labelling of the vertices of G' that identifies the connected components of G' correctly. Thus, since every isolated vertex of G'' is assigned a unique label in Line 8 of the algorithm, the labelling of the vertices of G'' obtained in Line 8 identifies the connected components of G'' correctly. We have to show that the labelling of G computed in Line 9 is correct.

So let $v, w \in G$, and assume first that v and w belong to the same connected component of G. Then there exists a path $P = (v = x_0, x_1, \dots, x_k = w)$ from v to w in G. It suffices to show that for every edge $\{x_i, x_{i+1}\} \in P$, $\gamma_G(x_i) = \gamma_G(x_{i+1})$ because then $\gamma_G(v) = \gamma_G(x_0) = \gamma_G(x_1) = \dots = \gamma_G(x_k) = \gamma_G(w)$. If vertices x_i and x_{i+1} belong to the same connected component of H, $\gamma_G(x_i) = \gamma_G(x_{i+1})$ because x_i and x_{i+1} receive their labels from the same vertex in G''. Otherwise let $x_i \in C_h$ and $x_{i+1} \in C_j$, $h \neq j$. Since edge $\{x_i, x_{i+1}\} \in G$, graph G'' contains edge $\{v_h, v_j\}$. Thus, vertices v_h and v_j belong to the same connected component of G'', so that $\gamma_{G''}(v_h) = \gamma_{G''}(v_j)$ and hence $\gamma_G(x_i) = \gamma_G(x_{i+1})$.

Now assume that $\gamma_G(v) = \gamma_G(w)$, and let $v \in C_h$ and $w \in C_j$. If $C_h = C_j$, there exists a path from v to w in $C_h \subseteq H \subseteq G$. If $C_h \neq C_j$, $\gamma_{G''}(v_h) = \gamma_G(v) = \gamma_G(w) = \gamma_{G''}(v_j)$. That is, vertices v_h and v_j belong to the same connected component of G''. In particular, there exists a path $P'' = (v_h = y_0, y_1, \dots, y_k = v_j)$ from v_h to v_j in G''. Let $j_0, \dots, j_k$ be indices so that for $0 \leq i \leq k$, vertex y_i represents component C_{j_i} of H. Since edge $\{y_i, y_{i+1}\} \in G''$, for $0 \leq i < k$, graph G contains edges $e_i = \{a_i, b_i\}$, $a_i \in C_{j_i}$ and $b_i \in C_{j_{i+1}}$. Since vertices b_i and a_{i+1} are in the same connected component $C_{j_{i+1}}$ of H, for $0 \leq i < k-1$, there exists a path P_{i+1} from b_i to a_{i+1} in $C_{j_{i+1}}$. Similarly, there exist paths P_0 and P_k from v to a_0 in C_{j_0} and from b_{k-1} to w in C_{j_k}. Hence, there exists a path $P = P_0 \circ e_0 \circ P_1 \circ e_1 \circ \dots \circ P_{k-1} \circ e_{k-1} \circ P_k$ from v to w in G, so that v and w belong to the same connected component of G. $\square$

If $|V| \leq M$, procedure ExternalConnectivity computes the connected components of G in $\mathcal{O}(\text{scan}(|V|+|E|))$ I/Os by invoking procedure SemiExternalConnectivity. Otherwise Lines 4–9 of the algorithm are executed, whose I/O-complexity we analyze next.

To find the smallest neighbor w_v of every vertex v, scan the edge set of G, replace every edge $\{v, w\} \in E$ with two directed edges (v, w) and (w, v), and sort the resulting set of directed edges lexicographically. The result is a collection of sorted adjacency lists of the vertices of G. Scan this collection of adjacency lists and select vertex w_v for every vertex $v \in G$ as the first vertex in the adjacency list of v. Computing the smallest neighbors of all vertices of G in this manner takes $\mathcal{O}(\text{sort}(|E|))$ I/Os. To produce the edge set of graph H, sort and scan the set of edges $\{v, w_v\}$, $v \in G$, to remove duplicates. This takes another $\mathcal{O}(\text{sort}(|V|))$ I/Os.

The most interesting part of the algorithm is the computation of the connected components of graph H because it has to be done without using procedure

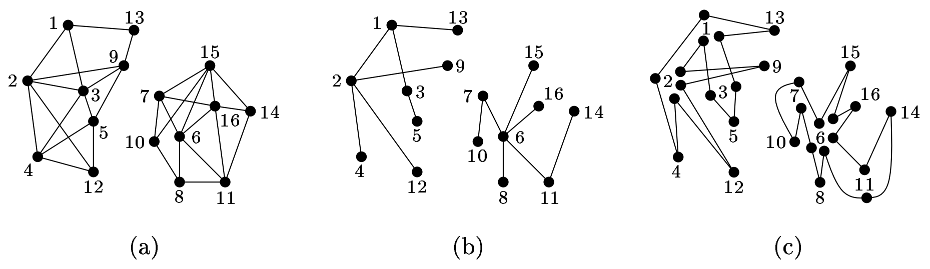

Figure 5.1
(a) A graph G with its vertices numbered. (b) The graph H induced by this numbering. (c) The graph H' whose connected components are cycles and represent the connected components of H. Every vertex in H' is labelled with the number of its corresponding vertex in H.

ExternalConnectivity. To do this, the algorithm makes use of the following fact.

Lemma 5.2. *The connected components of graph H are trees. That is, graph H is a forest.*

Proof. Assume for the sake of contradiction that graph H contains a cycle $C = (x_0, x_1, \ldots, x_k = x_0)$. Since graph H contains no parallel edges, $k \geq 3$. Since every vertex $v \in G$ has at most one incident edge $\{v, w_v\}$ in H, w.l.o.g. $x_{i+1} = w_{x_i}$, for $0 \leq i < k$. Then the existence of edge $\{x_{i-1}, x_i\}$, for $0 < i < k-1$, implies that $x_{i-1} > x_{i+1}$. Similarly, $x_{k-1} > x_1$. If k is even, this implies that $x_0 > x_2 > \cdots > x_k = x_0$, which leads to the desired contradiction. If k is odd, we arrive at a contradiction by observing that $x_0 > x_2 > \cdots > x_{k-1} > x_1 > x_3 > \cdots > x_k = x_0$. Hence, graph H contains no cycles, and all its connected components are trees. □

Using Lemma 5.2, the connected components of H can be found as follows: Apply the Euler tour technique to H, in order to transform each tree T in H into a cycle C_T (see Figure 5.1). Let H' be the resulting graph. Every vertex v' in a cycle C_T corresponds to a vertex v in T. During the construction of H' from H, vertex v' can easily be labelled with the ID of vertex v. Cycles C_T are the connected components of H', so that a labelling of the connected components of H can be obtained from a labelling of the connected components of H' as follows: Scan the set of vertex-label pairs $(v', \gamma_{H'}(v'))$ and replace each such pair with the pair $(v, \gamma_H(v) = \gamma_{H'}(v'))$, where v is the vertex in H represented by v'. Now sort and scan the resulting list of vertex-label pairs to remove duplicates.

To compute the connected components of H', a procedure similar to the list ranking algorithm from Section 2 can be used: If $|H'| \leq M$, load H' into main memory and compute its connected components using an efficient internal memory algorithm. Otherwise find a large independent set I of H' and remove the vertices in I from H', where removing a vertex v with incident edges $\{v, x\}$ and $\{v, y\}$ means to remove vertex v from the vertex set of H' and replace edges $\{v, x\}$ and $\{v, y\}$ with an edge $\{x, y\}$ connecting the neighbors of v. The removal of the vertices in I from H' then results in a collection of smaller cycles. Recursively find the connected components of this compressed graph and re-integrate the vertices in I, assigning to every vertex in I the component label of one of its neighbors. The details of this procedure are similar to those of Algorithm 2.2, so that it takes $\mathcal{O}(\text{sort}(|H'|)) = \mathcal{O}(\text{sort}(|V|))$ I/Os.

Given a labelling of the connected components of H, graph G' is now constructed in two phases: First scan the vertex set of G and create a list $V' = \{\gamma_H(v) : v \in G\}$. Sort and scan this list to remove duplicates. The result is the vertex set of G'. Now sort the vertices of G by their IDs and the edges of G by their first endpoints. Scan the two sorted lists to replace the first endpoint v of each edge $\{v, w\}$ with its component label $\gamma_H(v)$. Repeat this procedure, sorting the edges by their second endpoints to replace these endpoints with their component labels. Finally sort and scan the resulting list of edges $\{\gamma_H(v), \gamma_H(w)\}$ to remove duplicates and loops. This first phase of the construction of G' takes $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os and produces graph G' with its isolated vertices present (i.e., graph G'' in the proof of Lemma 5.1).

To remove all isolated vertices from G', scan the edge set of G' and append vertices v and w to a list X, for every edge $\{v, w\} \in E(G')$. Sort the vertex set of G' and list X and scan the two sorted lists to remove every vertex from $V(G')$ that does not appear in X. This takes another $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os. In total, the construction of G' from G takes $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os.

After recursively computing the connected components of G', the algorithm has to assign unique labels to the isolated vertices that were removed from G' and then derive a component labelling of G from the resulting labelling of graph G''. To label the isolated vertices of G'', sort the vertices of G' by their component labels and then scan the vertex set of G' and the set of isolated vertices to assign to each isolated vertex a label that has not been assigned to any other vertex. Since the ID of a vertex in G'' is in fact the label of a connected component in H, the resulting set of vertex-label pairs can be interpreted as pairs (γ_H, γ_G) mapping a component label in H to a component label in G. Now sort the vertices of G by their component labels in H, sort the list of pairs (γ_H, γ_G) by their first components, and finally scan both lists to replace the component label $\gamma_H(v)$ of every vertex $v \in G$ with the corresponding component label in G. This whole procedure takes $\mathcal{O}(\text{sort}(|V|+|E|))$ I/Os and derives a component labelling of G from the given component labellings of H and G'.

From this discussion we obtain the following recurrence describing the I/O-complexity of procedure ExternalConnectivity:

$$\mathcal{I}(|V|, |E|) = \begin{cases} \mathcal{O}(\text{scan}(|V| + |E|)) & \text{if } |V| \leq M \\ \mathcal{O}(\text{sort}(|V| + |E|)) + \mathcal{I}(|V(G')|, |E(G')|) & \text{if } |V| > M \end{cases}$$

Using this recurrence, we can show the following lemma.

Lemma 5.3. *The I/O-complexity of procedure* ExternalConnectivity *is* $\mathcal{I}(|V|, |E|) = \mathcal{O}(\text{sort}(|V|) + \text{sort}(|E|) \log_2(|V|/M))$.

Proof. If we can show that $|V(G')| \leq |V|/2$, the lemma follows from the above recurrence. To prove that the former is true, observe that every vertex in G' represents a connected component of H that contains at least two vertices. This is true because every vertex in G that is not isolated in G has at least one incident edge in H; an isolated vertex in G is also isolated in G' and is hence removed before recursively invoking procedure ExternalConnectivity on G'. Since the connected components of H define a partition of the vertex set of G, it now follows immediately that $|V(G')| \leq |V|/2$. □

The following theorem is an immediate consequence of Lemmas 5.1 and 5.3.

Theorem 5.4. *The connected components of an undirected graph* $G = (V, E)$ *can be computed in* $\mathcal{O}(\text{sort}(|V|) + \text{sort}(|E|) \log_2(|V|/M))$ *I/Os.*

An improved connectivity algorithm. In internal memory, the connected components of a graph can be computed in linear time using breadth-first search (BFS). Since the best known BFS-algorithm for undirected graphs is less efficient than procedure ExternalConnectivity, except for dense graphs, this idea does not directly lead to an improved connectivity algorithm. However, Munagala and Ranade [28] observed that the I/O-complexity of procedure ExternalConnectivity can be improved by using an I/O-efficient BFS-algorithm instead of procedure SemiExternalConnectivity to stop the recursion of procedure ExternalConnectivity. In particular, they present a BFS-algorithm for undirected graphs that takes $\mathcal{O}(|V| + \text{sort}(|E|))$ I/Os. We discuss this algorithm in Section 6.2. For $|V| \leq |E|/B$, the I/O-complexity of the algorithm is $\mathcal{O}(\text{sort}(|E|))$. In order to reduce the size of the vertex set of G to $|E|/B$, $\log_2(|V|B/|E|)$ recursive invocations of procedure External-Connectivity suffice, so that the improved algorithm takes $\mathcal{O}(\text{sort}(|V|) + \text{sort}(|E|) \log_2(|V|B/|E|))$ I/Os.

Theorem 5.5. *The connected components of an undirected graph* $G = (V, E)$ *can be computed in* $\mathcal{O}(\text{sort}(|V|) + \text{sort}(|E|) \log_2(|V|B/|E|))$ *I/Os.*

Remark. Munagala and Ranade improve the I/O-complexity of their connectivity algorithm even further, to $\mathcal{O}(\text{sort}(|V|) + \text{sort}(|E|) \log_2 \log_2(|V|B/|E|))$. The idea is to group the contraction steps into superphases. Each superphase achieves a contraction of the vertex set of G by a factor greater than two and takes $\mathcal{O}(\text{sort}(|E|))$ I/Os. To achieve the latter, the contraction steps in each superphase operate on a well-chosen subset of the edges of the graph. The interested reader may refer to [28] for details.

5.3 Biconnectivity

Tarjan and Vishkin [30] propose a parallel algorithm for computing the biconnected components of a graph $G = (V, E)$. The algorithm constructs an auxiliary graph H with $|E|$ vertices and $\mathcal{O}(|E|)$ edges so that the connected components of H correspond to the biconnected components of G and then computes the connected components of H. Chiang *et al.* [12] show that the construction of the auxiliary graph H can be carried out in $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os. This leads to the following corollary of Theorem 5.5. For details see [12, 30].

Theorem 5.6. *The biconnected components of an undirected graph $G = (V, E)$ can be computed in $\mathcal{O}(\text{sort}(|V|) + \text{sort}(|E|) \log_2 B)$ I/Os.*

5.4 Minimum Spanning Tree

Now let us turn to another problem that can be solved by refining the ideas from Section 5.2: computing a minimum spanning tree of a connected undirected graph $G = (V, E)$. A *spanning tree* of G is a tree $T = (V, E')$, $E' \subseteq E$. That is, tree T contains all vertices of G; its edge set is a subset of the edges of G. Given an assignment $\omega : E \to \mathbb{R}$ of weights to the edges of G, tree T is a *minimum spanning tree* (MST) of G if there is no spanning tree of G whose total edge weight is less than that of T.

The first step towards computing an MST of G is to observe that Algorithms 5.1 and 5.2 can easily be augmented to obtain procedures SEMIEXTERNALST and EXTERNALST that compute a spanning tree of G. The computed spanning tree is not necessarily a minimum spanning tree. The required modifications are the following:

In addition to relabelling the vertices in G, procedure SEMIEXTERNALST adds edge $\{v, w\}$ to the spanning tree whenever it finds that the endpoints v and w of the current edge $\{v, w\}$ are in different connected components.

Procedure EXTERNALST constructs a spanning tree T of G from graph H and the spanning tree T' produced by the recursive invocation of the algorithm on G'. The edge set of T contains all edges of graph H as well as one edge $\{v, w\}$ per edge $\{v,', w'\} \in T'$, where v and w are in the connected components of H represented by vertices v' and w'.

We leave the proof that procedure SEMIEXTERNALST computes a spanning tree of T as an exercise and show the following lemma.

Lemma 5.7. *Let T be the graph computed by procedure* EXTERNALST. *Then T is a spanning tree of G.*

Proof. We prove the lemma by induction on $|V|$. If $|V| \leq M$, graph T is computed using procedure SEMIEXTERNALST. Graph T is a spanning tree of G, by the correctness of procedure SEMIEXTERNALST. So assume that $|V| = k > M$ and that procedure EXTERNALST computes a spanning tree for every graph $G' = (V', E')$ with $|V'| < k$.

We have to show that the graph T computed for graph G is connected and does not contain cycles. So let v and w be two vertices of G, let $C_1, \dots, C_q$ be the connected components of graph H, and let $v \in C_h$ and $w \in C_j$. If $C_h = C_j$, there exists a path from v to w in $C_h \subseteq H \subseteq T$. If $C_h \neq C_j$, consider the graph T'

computed by recursively invoking procedure ExternalST on the compressed graph G'. By the induction hypothesis and because $|V(G')| < |V|$, graph T' is a spanning tree of G'. Hence, there exists a path $P' = (v_h = x_0, x_1, \dots, x_k = v_j)$ from v_h to v_j in T'. Let $j_0, \dots, j_k$ be indices so that vertex x_i represents component C_{j_i} of H, for $0 \le i \le k$. Since graph T' contains edges $\{x_i, x_{i+1}\}$, for $0 \le i < k$, graph T as constructed by procedure ExternalST contains an edge $e_i = \{a_i, b_i\}$, where $a_i \in C_{j_i}$ and $b_i \in C_{j_{i+1}}$, for $0 \le i < k$. Let P_0 be a path from v to a_0 in C_{j_0}, P_k be a path from b_{k-1} to w in C_{j_k}, and P_i be a path from b_{i-1} to a_i, for $0 < i < k$. Since $H \subseteq T$, the path $P = P_0 \circ e_0 \circ P_1 \circ e_1 \circ \cdots \circ P_{k-1} \circ e_{k-1} \circ P_k$ is a path from v to w in T. As this is true for every pair of vertices $v, w \in G$, graph T is connected.

Now assume for the sake of contradiction that graph T contains a cycle $C = (x_0, x_1, \dots, x_k = x_0)$. Cycle C can be split into maximal subpaths $P_1, \dots, P_l$ so that the vertices of each subpath P_i belong to the same connected component C_{j_i} of H. By Lemma 5.2, the connected components of H are trees, so that the partition of cycle C contains at least two paths P_1 and P_2. An edge in C connecting two vertices in different subpaths P_i and P_{i+1} (or P_l and P_1) has its two endpoints in C_{j_i} and $C_{j_{i+1}}$ (or C_{j_l} and C_{j_1}). By the construction of T and since $C \subseteq T$, this implies that graph T' contains a cycle $C' = (v_{j_1}, v_{j_2}, \dots, v_{j_l}, v_{j_1})$. However, by the induction hypothesis, graph T' is a tree, and hence does not contain any cycles. This leads to the desired contradiction, so that T is a tree.

$\square$

Only a few modifications to procedures SemiExternalST and ExternalST are required to make them compute *minimum* spanning trees of their input graphs. We describe these modifications below and refer to the resulting algorithms as procedures SemiExternalMST and ExternalMST.

Instead of inspecting the edges of G in an arbitrary order, procedure SemiExternalMST inspects the edges sorted by increasing weights. This increases the I/O-complexity of the algorithm to $\mathcal{O}(\text{scan}(|V|) + \text{sort}(|E|))$ because the edges have to be sorted before scanning the edge set. With this modification, procedure SemiExternalMST becomes a semi-external version of Kruskal's algorithm [13, Section 24.2] and hence computes an MST of G.

Procedure ExternalMST differs from procedure ExternalST in a number of places; but all modifications are simple:

(1) During the construction of H from G, edge $\{v, w_v\}$ is chosen as the minimum-weight edge incident to v instead of the edge connecting v to its smallest neighbor. It is easy to verify that this modification maintains the invariant that H is a forest.

(2) Every edge $\{v', w'\} \in G'$ represents a set of edges in G between the two connected components of H represented by v' and w'. The weight of edge $\{v', w'\}$ is chosen as the minimum weight of all edges in this set.

(3) When adding an edge $\{v, w\}$ to T for an edge $\{v', w'\} \in T$, then $\{v, w\}$ is chosen as an edge of minimum weight so that vertices v and w belong to the connected components of H represented by v' and w'. In particular, edges $\{v, w\}$ and $\{v', w'\}$ have the same weight.

We leave it as an exercise to verify that these modifications do not increase the I/O-complexity of procedure ExternalMST. By Lemma 5.2, the graph T computed by the algorithm is a spanning tree of G. Next we show that it is a minimum spanning tree.

Lemma 5.8. *The graph T computed by procedure* ExternalMST *is a minimum spanning tree of G.*

Proof. We prove the lemma by induction on $|V|$. If $|V| \leq M$, the correctness of procedure ExternalMST follows from that of procedure SemiExternalMST because it uses this procedure to compute T. So assume that $|V| = k > M$ and that procedure ExternalMST computes an MST for any graph with less than k vertices.

First we show that graph G has an MST T so that $H \subseteq T$. Assume the contrary, and let T be an MST of G that contains a maximal number of edges of H. Since $H \not\subseteq T$, there exists an edge $\{v, w_v\} \in H$ that is not in T. Adding edge $\{v, w_v\}$ to T creates a cycle C in T. Since graph H is a forest, cycle C contains an edge $\{x, y\} \notin H$. Assume w.l.o.g. that y is on the path from x to v in T, and choose edge $\{x, y\}$ so that the path $P = (y = x_0, x_1, \ldots, x_k = v)$ from y to v in T contains only edges of H. Since $\{v, w_v\}$ is the edge of minimum weight incident to v chosen during the construction of graph H, edge $\{x_{k-1}, x_k\}$ has weight at least that of edge $\{v, w_v\}$. Moreover, edge $\{x_{k-1}, x_k\}$ can only be chosen by its two endpoints as a minimum weight edge to be added to H, and vertex $x_k = v$ chose edge $\{v, w_v\}$. Hence, edge $\{x_{k-1}, x_k\}$ is the minimum weight edge chosen for vertex x_{k-1}. Using induction, we obtain that for $0 \leq i < k$, edge $\{x_i, x_{i+1}\}$ is the minimum weight edge chosen for vertex x_i and that the weight of this edge is no less than that of edge $\{v, w_v\}$. Since edge $\{x, y\}$ is incident to vertex $y = x_0$, its weight is no less than that of edge $\{x_0, x_1\}$, which is no less than that of edge $\{v, w_v\}$. Thus, replacing edge $\{x, y\}$ with edge $\{v, w_v\}$ in T produces a spanning tree T_0 of weight no more than that of T and containing one more edge of H than T. This contradicts the choice of T, so that $H \subseteq T$.

It remains to show that the algorithm adds the correct edges to H in order to construct tree T. So assume that T is not an MST, and let T_0 be an MST of G so that $H \subseteq T_0$. Contracting every connected component of H into a single vertex transforms T into the tree T' computed for G' by the recursive invocation of procedure ExternalMST. Tree T_0 is transformed into another spanning tree T_0' of G'. All edges in T_0' and T' have the same weights as their corresponding edges in T_0 and T, and the edges in H are shared by T_0 and T. Hence, the difference between the weights of T_0 and T is the same as the difference between the weights of T_0' and T'. By the induction hypothesis, T' is an MST of G'. Thus, T_0' has a weight no less than that of T', so that the weight of T_0 is at least that of T. Hence, T is an MST of G. □

Since the I/O-complexity of procedure ExternalMST is the same as that of procedure ExternalConnectivity, the following theorem now follows from Theorem 5.4 and Lemma 5.8.

Theorem 5.9. *A minimum spanning tree of a connected undirected graph $G = (V, E)$ can be computed in $\mathcal{O}(\text{sort}(|V|) + \text{sort}(|E|)\log_2(|V|/M))$ I/Os.*

Now it would be nice if the same trick as for the connectivity algorithm could be applied to stop the recursion in procedure EXTERNALMST already after $\log_2(|V|B/|E|)$ recursive calls. That is, we are looking for an algorithm that computes a minimum spanning tree of a graph $G = (V, E)$ in $\mathcal{O}(|V| + \text{sort}(|E|))$ I/Os. The BFS-algorithm of Munagala and Ranade [28] cannot be used because a BFS-tree of G is most likely not an MST.

Arge *et al.* [4] present an I/O-efficient version of Prim's algorithm [13, Section 24.2] that computes an MST of G in the desired number of I/Os. As Prim's algorithm, the algorithm of [4] maintains the invariant that the current set of edges defines a spanning tree of a subset of the vertices of G and that this spanning tree is a subgraph of a minimum spanning tree of G. To extend the spanning tree, the edge of lowest weight connecting a vertex in the spanning tree to a vertex not in the spanning tree is added to the tree. This operation is repeated until a minimum spanning tree of G is obtained.

More precisely, the algorithm starts by choosing one vertex r to be in the spanning tree, while all other vertices are not in the spanning tree. Then the adjacency list of r is retrieved, and for every edge $\{r, x\}$ incident to r, an edge (r, x) is inserted into a priority queue Q storing edges (v, w) so that $v \in T$. The priority of an edge $(v, w) \in Q$ is the same as the weight of edge $\{v, w\}$ in G.

To find the next edge to be added to the current spanning tree T, the edge (u, v) of lowest weight is retrieved from Q. If this edge connects two vertices in the spanning tree, it is discarded, and the next edge is retrieved. Otherwise edge $\{u, v\}$ is added to T. Since $u \in T$, vertex v was not in T before adding edge $\{u, v\}$ to T. To update Q, the adjacency list of v is retrieved, and for every edge $\{v, w\}$, $u \neq w$, incident to v, an edge (v, w) is inserted into priority queue Q.

The correctness of the algorithm follows from that of Prim's algorithm because it maintains the invariant that priority queue Q stores all edges connecting vertices in T to vertices not in T. The main difficulty is to find an I/O-efficient method to test whether an inspected edge connects two vertices in T or a vertex in T with a vertex not in T. Under the assumption that no two edges have equal weight[4] this test can be carried out using priority queue Q: Observe that if the two endpoints u and v of an inspected edge (u, v) are in T, but $\{u, v\} \notin T$, vertex u has inserted edge (u, v) into Q, and vertex v has inserted edge (v, u) into Q. Hence, if u and v are both in T, and the current DELETEMIN operation retrieves edge (u, v), the edge retrieved by the next DELETEMIN operation is (v, u). Thus, it suffices to perform two DELETEMIN operations. If these two operations retrieve two edges with the same endpoints, both edges are discarded. Otherwise the first edge is added to the spanning tree, and the second edge is re-inserted into Q.

An important detail to be observed is the fact that when edge $\{u, v\}$ is added to the spanning tree, edge (v, u) is excluded from the set of edges in the adjacency

[4] This can easily be achieved by defining new edge weights $\omega'(e) = (\omega(e), e)$ and taking the lexicographical order as the natural order on these edge weights.

list of v that are inserted into Q. This is important because edge (u, v) has just been retrieved from Q, so that the above test would fail when edge (v, u) is retrieved by a subsequent DELETEMIN operation.

To analyze the I/O-complexity of the algorithm, observe that it takes $\mathcal{O}(|V|+\text{scan}(|E|))$ I/Os to retrieve the adjacency lists of all vertices of G. Besides retrieving the adjacency lists, the algorithm performs $\mathcal{O}(|E|)$ priority queue operations: $\mathcal{O}(|E|)$ INSERT operations are performed to insert every edge of G into Q for the first time. All other priority queue operations can be grouped into sequences of either two DELETEMIN operations or two DELETEMIN operations followed by the re-insertion of the edge retrieved by the last DELETEMIN operation. Each such sequence of priority queue operations reduces the number of edges stored in Q by at least one, so that at most $\mathcal{O}(|E|)$ such sequences are executed. Since each sequence has length at most three, the total number of priority queue operations is $\mathcal{O}(|E|)$, which take $\mathcal{O}(\text{sort}(|E|))$ I/Os to be performed. Hence, the total I/O-complexity of the algorithm is $\mathcal{O}(|V| + \text{sort}(|E|))$.

Using the above algorithm instead of procedure SEMIEXTERNALMST in procedure EXTERNALMST, the recursion can stop after $\log_2(|V|B/|E|)$ recursive calls, so that we obtain the following result.

Theorem 5.10. *A minimum spanning tree of a connected undirected graph* $G = (V, E)$ *can be computed in* $\mathcal{O}(\text{sort}(|V|) + \text{sort}(|E|) \log_2(|V|B/|E|))$ *I/Os.*

Remark. Similar to the connectivity algorithm of [28], the complexity of the MST-algorithm can be reduced to $\mathcal{O}(\text{sort}(|V|) + \text{sort}(|E|) \log_2 \log_2(|V|B/|E|))$. This improvement is achieved using essentially the same approach as in [28]; but a number of interesting new ideas are used. The interested reader may refer to [4] for details.

5.5 Graph Contraction and Sparse Graphs

Observe that the algorithms in Sections 5.2 and 5.4 are optimal in the number of vertices in the graph, but not in the number of edges. This is due to the fact that graph G' has at most half as many vertices as graph G, while no sufficiently good upper bounds on the number of edges in G' can be given. However, if graph G is sparse, the I/O-complexity of procedures EXTERNAL-CONNECTIVITY and EXTERNALMST is reduced to $\mathcal{O}(\text{sort}(|V|))$. In particular, we say that graph G is *sparse* if $|E(H)| = \mathcal{O}(|V(H)|)$ for every graph H that can be obtained from G through a series of edge contractions. Important classes of sparse graphs include planar graphs, grid graphs, and graphs of bounded treewidth. Since graph G' is obtained from graph G through a series of edge contractions, the sparseness of G implies that $|E(G')| = \mathcal{O}(|V(G')|)$, so that the I/O-complexity of procedures EXTERNALCONNECTIVITY and EXTERNALMST is now $\mathcal{O}(\text{sort}(|V|) + \text{sort}(|V|/2) + \text{sort}(|V|/4) + \dots) = \mathcal{O}(\text{sort}(|V|))$. Hence, we obtain the following result.

Theorem 5.11. *For every sparse graph* $G = (V, E)$*, the connected components or a minimum spanning tree of* G *can be computed in* $\mathcal{O}(\text{sort}(|V|))$ *I/Os. The latter exists only if* G *is connected.*

6 Breadth-First Search and Depth-First Search

Breadth-first search (BFS) and depth-first search (DFS) are probably among the most fundamental primitives used to study the structure of a given graph. Sequential algorithms for finding the biconnected components [29] and triconnected components [18] of a graph and the first linear-time algorithm for planarity testing [19] are based on depth-first search. Breadth-first search can be seen as an unweighted version of the single source shortest path problem and besides that has been employed for example in algorithms for computing planar separators [23, and many more]. The popularity of BFS and DFS in sequential graph algorithms is not surprising, as these procedures can be carried out in linear time using extremely simple algorithms; yet their output provides valuable information about the structure of the graph. If it is possible to design I/O-efficient algorithms for BFS and DFS, then there is hope to obtain I/O-efficient versions of many sequential graph algorithms based on BFS and DFS. Unfortunately no generally I/O-efficient BFS or DFS-algorithms are known. Before discussing what can be done, let us see what we can establish using rather simple observations.

First, we should not hope to obtain a linear-I/O algorithm for either BFS or DFS because the list ranking problem can be solved by performing BFS or DFS from the head of the list. That is, BFS and DFS require $\Omega(\text{perm}(N))$ I/Os. We state this as a corollary in Section 11, which deals with lower bounds.

Second, the internal memory algorithms are not I/O-efficient. In particular, they perform $\mathcal{O}(|V| + |E|)$ I/Os in the worst case: At least one I/O is required to access the adjacency list of each vertex. Every edge to be explored requires to check whether the other endpoint of the edge has been visited before. This requires one I/O in the worst case, so that the algorithm performs one I/O per edge.

6.1 Directed BFS and DFS

The first I/O-efficient algorithms for BFS and DFS we discuss work for directed graphs. While no better DFS-algorithm is known for undirected graphs, simpler and faster BFS-algorithms for undirected graphs exist. We discuss these algorithms in Sections 6.2 and 6.3.

The buffered repository tree. The buffered repository tree (BRT) [11] is the key data structure used to obtain I/O-efficient algorithms for BFS and DFS in directed graphs. A BRT stores key-value pairs (k, v) and supports two operations: $\textsc{Insert}((k, v))$ and $\textsc{Extract}(k)$. Operation $\textsc{Insert}$ inserts the given key-value pair into the BRT and takes $\mathcal{O}\left(\frac{1}{B} \log_2 \frac{N}{B}\right)$ I/Os. Operation $\textsc{Extract}$ removes all key-value pairs with key k from the BRT and returns them. This operation takes $\mathcal{O}(\log_2(N/B))$ I/Os. The I/O-bounds of both operations are amortized.

The BRT is a $(2, 4)$-tree T that stores blocks of key-value pairs at its leaves, sorted by increasing keys. Every internal node of T has a buffer of size B. The root of T is held in main memory. All other nodes are stored on disk.

An INSERT operation inserts the new pair into the root buffer. If there is room for the new pair in the root buffer, this completes the operation and does not incur any I/Os. Otherwise the root buffer is emptied after inserting the new pair. To do this, the elements in the buffer are distributed to the appropriate children of the root and inserted into their buffers. This takes $\mathcal{O}(1)$ I/Os. But it may also cause the buffers of some of the children to overflow. If this happens, these buffers are emptied recursively. Once this recursive buffer-emptying process reaches the leaf level, it may be necessary to rebalance the tree. We discuss rebalancing below.

An EXTRACT operation traverses the whole subtree of T between the two paths to the leftmost and rightmost leaves of T storing elements with key k, including these two paths. At every visited leaf, the elements with key k are extracted. At every visited internal node, the buffer of the node is inspected, and all elements with key k are extracted. Then the empty leaves and all their ancestors having only empty leaves as descendants are removed from T, and T is rebalanced.

The following two lemmas state the I/O-complexities of INSERT and EXTRACT operations if the I/Os spent on rebalancing T are ignored. We analyze the cost of rebalancing below.

Lemma 6.1. *An* INSERT *operation on a BRT that stores N elements takes $\mathcal{O}\left(\frac{1}{B}\log_2\frac{N}{B}\right)$ I/Os amortized, excluding the I/Os required for rebalancing.*

Proof. Since the I/Os required for rebalancing are excluded from the analysis, it suffices to observe that the height of a BRT storing N elements is $\mathcal{O}(\log_2(N/B))$ and that emptying a buffer of size $X \geq B$ takes $\mathcal{O}(X/B)$ I/Os. Thus, the cost of the buffer emptying operation can be charged to the X elements in the buffer, charging every element for $\mathcal{O}(1/B)$ I/Os. Every inserted element is charged for $\mathcal{O}(1/B)$ I/Os per level, so that the I/O-bound follows. □

Lemma 6.2. *An* EXTRACT *operation on a BRT that stores N elements takes $\mathcal{O}\left(\log_2\frac{N}{B}+\frac{K}{B}\right)$ I/Os, excluding the I/Os required for rebalancing. K denotes the number of reported key-value pairs.*

Proof. An EXTRACT operation traverses $\mathcal{O}\left(\log_2\frac{N}{B}+\frac{K}{B}\right)$ nodes in the BRT: $\mathcal{O}(\log_2(N/B))$ nodes on the leftmost and rightmost paths bounding the range of elements with key k, and $\mathcal{O}(K/B)$ nodes between those paths. It is easy to see that visiting a single node costs $\mathcal{O}(1)$ I/Os, so that the I/O-bound follows. □

In order to finish the analysis of the buffered repository tree, we have to count the I/Os spent on rebalancing T. The rebalancing after an INSERT operation is done in the same manner as on a buffer tree [2] (see the chapter by Lars Arge). The rebalancing after an EXTRACT operation has to be done more carefully because it seems difficult to rebalance T after the whole subtree of extracted elements has been removed. Therefore, instead of removing all leaves in the subtree immediately, the leaves that become empty after an EXTRACT operation are marked for deletion. Then the marked leaves are deleted one by one, and the tree is rebalanced after every deletion. Since the creation or deletion of a leaf is triggered by the insertion or deletion of $\Omega(B)$ elements, the total number of leaf creations and deletions is $\mathcal{O}(N/B)$. As shown in [20], this implies that the total number of node splits, merges and fusions is bounded by $\mathcal{O}(N/B)$. Since each such operation can be performed in $\mathcal{O}(1)$ I/Os, the total number of I/Os spent on rebalancing T is $\mathcal{O}(N/B)$, and we obtain the following lemma.

Lemma 6.3. *The number of I/Os spent on rebalancing an initially empty BRT during a sequence of N* Insert *and* Extract *operations is $\mathcal{O}(N/B)$.*

Now the following theorem is an immediate consequence of Lemmas 6.1, 6.2, and 6.3, after observing that the $\mathcal{O}(K/B)$ I/Os spent by an Extract operation on reporting K key-value pairs can be charged to the K Insert operations that inserted the reported key-value pairs into T. This does not increate the amortized I/O-complexity of Insert operations by more than a constant factor.

Theorem 6.4. *An initially empty BRT supports* Insert *and* Extract *operations in $\mathcal{O}\left(\frac{1}{B}\log_2\frac{N}{B}\right)$ and $\mathcal{O}(\log_2(N/B))$ I/Os amortized, where N is the total number of* Insert *operations performed on T.*

Directed DFS. Having the BRT at our disposal, we can now proceed to the discussion of an I/O-efficient DFS-algorithm for directed graphs by Buchsbaum *et al.* [11]. The algorithm proceeds in the same manner as the internal memory algorithm: It maintains a stack storing the vertices on the path from the source s of the search to the current vertex v in the constructed DFS-tree. When visiting v, it explores the previously unexplored out-edges of v and tests whether the other endpoint w of such an edge (v, w) has been visited before. If not, v is declared to be w's parent in the constructed DFS-tree, w is pushed on the stack, and the same procedure is applied to w. If w has been visited before, the next out-edge of v is explored. If no unexplored out-edges remain, vertex v is removed from the stack, and the procedure backtracks to v's parent.

As pointed out earlier, this algorithm spends one I/O per vertex and one I/O per edge. In general, it is not known how to amend the former; but the following solution reduces the amortized cost per edge to $\mathcal{O}\left(\frac{1}{B}\log_2|V|\right)$ I/Os, at the expense of paying $\mathcal{O}(\log_2|V|)$ I/Os per vertex, which results in a DFS-algorithm that takes $\mathcal{O}((|V| + |E|/B)\log_2|V|)$ I/Os.

The algorithm makes use of the following data structures:

- A BRT T storing edges of G. Each edge has its source vertex as its key. Tree T is initially empty.
- A priority queue $P(v)$ per vertex $v \in G$, which stores the out-edges of v that have not been explored yet and whose other endpoints have not been visited before the last visit to v.

An important invariant maintained by the algorithm is that at any time, for any vertex v, the edges that are stored in $P(v)$ and are not stored in T are the edges from v to unvisited vertices. When vertex v is visited either for the first time or by backtracking from a descendant of v, an EXTRACT operation is performed on T to extract all edges with key v. These edges are deleted from $P(v)$ using DELETE operations. After that, priority queue $P(v)$ stores only edges from v to unvisited out-neighbors. If $P(v)$ is empty, v has no unvisited out-neighbors left, and the search backtracks. Otherwise the next edge to be explored is extracted using a DELETEMIN operation. Let this edge be (v, w). Then vertex w is pushed on the stack, the set of in-edges of w are retrieved, and every edge (x, w) in this set is inserted into T with key x. This maintains the invariant for every in-neighbor x of w and prevents the algorithm from exploring edge (x, w) when vertex x is visited.

The correctness of the algorithm is obvious, as it explores an edge if and only if the other endpoint of the edge has not been visited before. We split the analysis of the I/O-complexity of the algorithm into I/Os spent on updates of the BRT, priority queue operations and accessing adjacency lists.

Accessing the adjacency lists of all vertices of G takes $\mathcal{O}(|V| + |E|/B)$ I/Os because the adjacency list of every vertex is accessed exactly once. The number of priority queue operations performed by the algorithm is $\mathcal{O}(|E|)$: Initially, every edge (v, w) of G is inserted into exactly one priority queue, namely $P(v)$. After this initialization, only DELETEMIN and DELETE operations are performed on any priority queue, so that only $|E|$ of these operations can be performed before all priority queues are empty. Hence, using buffer trees [2] to implement the priority queues, the algorithm would spend $\mathcal{O}(\text{sort}(|E|))$ I/Os on all priority queue operations it performs, if there were room to keep a buffer of size B per priority queue in main memory. However, there are $|V|$ different priority queues, and in general we have to assume that $|V|B > M$. Therefore, the algorithm creates a buffer of size B only for the priority queue $P(v)$ of the current vertex v. Before making another vertex the active vertex, the buffer of priority queue $P(v)$ is emptied, even if it contains only few elements. This costs $\mathcal{O}(1)$ I/Os per visit to vertex v. Fortunately the DFS-algorithm performs an inorder traversal of the constructed DFS-tree, so that the number of visits to different vertices is $\mathcal{O}(|V|)$. Hence, the total number of I/Os spent on priority queue operations is $\mathcal{O}(|V| + \text{sort}(|E|))$.

Finally, the algorithm performs $\mathcal{O}(|E|)$ INSERT operations and $\mathcal{O}(|V|)$ EXTRACT operations on the BRT. Each INSERT operation takes $\mathcal{O}\left(\frac{1}{B} \log_2 \frac{|E|}{B}\right) = \mathcal{O}\left(\frac{1}{B} \log_2 |V|\right)$ I/Os amortized. Each EXTRACT operation takes $\mathcal{O}(\log_2 |V|)$ I/Os amortized. Hence, the total number of I/Os spent on updating the BRT is $\mathcal{O}((|V| + |E|/B) \log_2 |V|)$. We obtain the following result.

Theorem 6.5. *A DFS-tree of a directed graph $G = (V, E)$ can be computed in $\mathcal{O}((|V| + |E|/B) \log_2 |V|)$ I/Os.*

Directed BFS. In order to obtain an I/O-efficient BFS-algorithm for directed graphs, it suffices to modify the above algorithm so that it uses a queue instead of a stack to determine the order in which the vertices of G are visited. That is, when visiting a vertex v, the out-edges of v leading to visited neighbors of v are extracted from T and deleted from $P(v)$. The remaining edges in $P(v)$ are retrieved using a series of DELETEMIN operations. For every retrieved edge (v, w), vertex v is declared to be the parent of w, vertex w is appended to the end of the queue, and all in-edges of w are inserted into the BRT. Once priority queue $P(v)$ has been emptied in this manner, the next vertex to be visited is retrieved from the head of the queue.

The analysis of the algorithm is the same as for DFS after observing that the number of visits to different vertices is again $\mathcal{O}(N)$ because now every vertex is visited exactly once. (The algorithm does not backtrack.) Hence, we obtain the following result.

Theorem 6.6. *A BFS-tree of a directed graph $G = (V, E)$ can be computed in $O((|V| + |E|/B) \log_2 |V|)$ I/Os.*

Remark. We leave it as an exercise to verify that for BFS, the use of priority queues $P(v)$, $v \in V$, can be avoided altogether because every vertex is visited exactly once.

6.2 Undirected BFS

The algorithms for BFS and DFS in directed graphs follow the framework of the internal memory algorithms for these problems, but spend a lot of effort on efficiently maintaining the set of vertices they have visited so far. For BFS in undirected graphs, Munagala and Ranade [28] exploit the particularly simple structure of BFS-trees of these graphs in order to design a BFS-algorithm that takes $\mathcal{O}(|V| + \text{sort}(|E|))$ I/Os.

This "particularly simple structure" of BFS-trees of undirected graphs is characterized as follows: Let v be a vertex at distance d from the root of a BFS-tree of the graph. Then all neighbors of v are at distance $d-1$, d, or $d+1$ from the root. Hence, when the algorithm visits vertex v, only the nodes at distances $d-1$ and d have to be inspected to find out which neighbors of v have been visited before. All other nodes are either children of v or of another node at level d in the BFS-tree. This eliminates the need for a complicated data structure to keep track of the vertices the algorithm has already visited.

Given the root r of the BFS-tree T to be computed, the algorithm computes a partition of the vertices of G into disjoint sets $L(0), L(1), \dots$ so that the vertices in set $L(i)$ are at distance i from r. That is, set $L(0)$ contains only the root r of T, set $L(1)$ contains all neighbors of r, and so on. We call sets $L(0), L(1), \dots$ the *levels* of tree T. The algorithm computes these levels iteratively, starting with $L(0) = \{r\}$. Given levels $L(0), \dots, L(i)$, the next level $L(i + 1)$ is computed as the difference between the set of neighbors of all vertices in $L(i)$ and the union of sets $L(i-1)$ and $L(i)$. This process is repeated until the most recently computed level $L(i)$ is empty. The pseudo-code of the algorithm is shown in Algorithm 6.1.

The correctness of this procedure follows from the above observation. To analyze the I/O-complexity of the algorithm, we bound the number of I/Os spent on accessing the adjacency lists of the vertices in G and the number of

```
Procedure UndirectedBFS
1: $L(-1) \leftarrow \emptyset$
2: $L(0) \leftarrow \{r\}$
3: $i \leftarrow 0$
4: while $L(i) \neq \emptyset$ do
5:     Let $X(i)$ be the union of the adjacency lists of all vertices in $L(i)$.
6:     Remove duplicates from $X(i)$.
7:     Remove all vertices in $L(i-1) \cup L(i)$ from $X(i)$.
8:     $L(i+1) \leftarrow X(i)$
9:     $i \leftarrow i+1$
10: end while
```

Algorithm 6.1
An I/O-efficient BFS-algorithm for undirected graphs.

I/Os spent on computing $L(i+1)$ from sets $L(i-1)$, $L(i)$ and $X(i)$. The number of I/Os spent on accessing adjacency lists is easily bounded by $\mathcal{O}(|V|+\text{scan}(|E|))$.

The computation of set $L(i+1)$ from sets $L(i-1)$, $L(i)$ and $X(i)$ requires sorting $L(i-1)$, $L(i)$ and $X(i)$. Once these lists are sorted, a single scan of these lists is sufficient to remove duplicates as well as all elements in $L(i-1) \cup L(i)$ from $X(i)$. Since sets $L(0), L(1), \ldots$ form a partition of the vertex set of G into disjoint sets, the total size of sets $L(0), L(1), \ldots$ is $|V|$. Each set $L(i)$ is involved in the computation of sets $L(i+1)$ and $L(i+2)$, so that the total number of I/Os spent on sorting and scanning sets $L(0), L(1), \ldots$ is $\mathcal{O}(\text{sort}(|V|)) = \mathcal{O}(\text{sort}(|E|))$. The total size of all sets $X(0), X(1), \ldots$ is $\mathcal{O}(|E|)$. To see this, observe that a vertex v is added to a set $X(i)$ because of an edge $\{u, v\}$ incident to v and so that $u \in L(i)$. Every edge causes each of its endpoints to be inserted into exactly one set $X(i)$, so that the total size of sets $X(0), X(1), \ldots$ is $2|E|$. Therefore the number of I/Os spent on sorting and scanning sets $X(0), X(1), \ldots$ is $\mathcal{O}(\text{sort}(|E|))$. This proves the following result.

Theorem 6.7. *A BFS-tree of an undirected graph $G = (V, E)$ can be computed in $\mathcal{O}(|V| + \text{sort}(|E|))$ I/Os.*

Remark. The BFS-algorithm as described in Algorithm 6.1 only computes the distance of every vertex from the root r of the BFS-tree. In order to make the algorithm compute the parent of each vertex in the BFS-tree, observe that a vertex ends up in $L(i+1)$ because it is in $X(i)$, but not in $L(i-1)$ or $L(i)$. A vertex v is in $X(i)$ because there is a vertex in $L(i)$ that is adjacent to v. Hence, instead of adding only vertex v to $X(i)$, a pair (v, u) can be added to $X(i)$, where u is the vertex in $L(i)$ that caused this copy of v to be inserted into $X(i)$. For every pair (v, u) that remains in $L(i+1)$ after removing duplicate pairs with the same first component, vertex u is a vertex in $L(i)$ adjacent to v, so that it can be made the parent of v in T.

6.3 A Faster Undirected BFS-Algorithm

While procedure UndirectedBFS is efficient for dense graphs, i.e., for graph with $|E| = \Omega(B|V|)$, it is no more efficient than the internal memory algorithm for graphs with $|E| = \mathcal{O}(|V|)$. In particular, the algorithm spends $\mathcal{O}(|V|)$ I/Os in this case, while the lower bound for BFS is only $\Omega(\text{perm}(|V|))$. In the last few years, one of the main challenges has been to develop BFS-algorithms that perform well on sparse graphs. A number of I/O-optimal algorithms for special classes of sparse graphs have been developed [4, 6, 24, 25]; but $\Omega(|V|)$ I/Os seemed to be a lower bound for BFS if no additional structural information about the graph is available.

Mehlhorn and Meyer [27] disproved this conjecture and made a major step towards closing the gap between the lower and upper bounds for BFS by developing a BFS-algorithm that takes $\mathcal{O}\left(\sqrt{|V||E|/B} + \text{sort}(|E|)\right)$ I/Os. For sparse graphs, for example, the algorithm takes $\mathcal{O}\left(|V|/\sqrt{B}\right)$ I/Os as opposed to $\mathcal{O}(|V|)$ I/Os spent by procedure UndirectedBFS.

We first discuss a randomized version of the algorithm because it provides the right intuition. Given the randomized algorithm, a simple observation suffices to make the algorithm deterministic.

The idea of the algorithm is to group the vertices of G into disjoint clusters of small diameter and then run procedure UndirectedBFS with a few modifications. First the algorithm makes sure that the adjacency lists of all vertices in the same cluster are stored consecutively. We refer to such a concatenation of adjacency lists as the *file* of the respective cluster. Whenever a vertex is first discovered, the algorithm does not only retrieve the adjacency list of the discovered vertex, but the whole file of the cluster containing that vertex. Thus, if the number of clusters is much smaller than the number of vertices, the number of random accesses spent on loading adjacency lists is much smaller than $|V|$. On the other hand, by incorporating all edges in a file into the computation already when the first vertex in the cluster is discovered, many edges may be involved in the computation of more than one level of the BFS-tree, which increases the number of I/Os spent on computing the levels from the retrieved files. That is, the algorithm trades off random accesses against spending more I/Os to perform the actual computation of the algorithm. As we will see, the trade-off balances at the above I/O-complexity. This trade-off is also the reason why it seems that this idea cannot be pushed further to obtain a BFS-algorithm that takes $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os.

The algorithm proceeds in two stages. The first stage forms clusters of small diameter. The second stage applies procedure UndirectedBFS after grouping the adjacency lists into files.

Forming clusters. First we describe the randomized clustering algorithm, as it provides some intuition about how the algorithm works and how a parameter μ to be specified later decreases the I/O-complexity of one part of the algorithm, while increasing the complexity of the other part of the algorithm.

So let $0 < \mu < 1$. Then the algorithm chooses a subset $V' \subseteq V$ of vertices by flipping a coin for every vertex in $V \setminus \{r\}$, where r is the root of the BFS tree to be computed. The coin comes up head with probability μ. Vertex v is included in set V' if its coin comes up head. Vertex r is always included in set V'. The vertices in V' are called *masters*, each being the center of a separate cluster. That is, the number of clusters formed by the algorithm is $|V'|$. Let the vertices in V' be $r = r_1, \ldots, r_q$. Then vertex r_i is the master of cluster C_i.

Observation 6.1. *The expected size of vertex set V' is $E[|V'|] \leq 1 + \mu|V|$.*

The clusters are now formed by running procedure UNDIRECTEDBFS from all masters simultaneously. That is, level $L(0)$ contains all masters. Then the algorithm is run as before until all vertices of G are discovered. Now observe that the algorithm assigns a parent to every vertex except to those in level $L(0)$. Hence, every vertex is a descendant of exactly one master in $L(0)$. Cluster C_i consists of all vertices having some vertex $r_i \in V'$ as an ancestor. The following lemma is the key to the efficiency of the algorithm.

Lemma 6.8. *The expected diameter of any cluster C_i is $2/\mu$.*

Proof. Consider any path $P = (r = x_k, x_{k-1}, \ldots, x_1, v)$ from r to a vertex $v \in C_i$. Since G is connected, path P exists. This guarantees that every vertex will be "captured" by some master. Hence, there is no vertex in G that is not contained in any cluster. Now let $1 \leq j \leq k$ be the smallest index so that x_j is a master. Since every vertex is chosen to be a master with probability μ, $E[j] = 1/\mu$. Hence, the expected distance of vertex v from the master of cluster C_i is at most $1/\mu$. Since this is true for any vertex in C_i, the lemma follows. □

We conclude the discussion of this first part of the algorithm with the analysis of its I/O-complexity.

Lemma 6.9. *A partition of the vertex set of a graph $G = (V, E)$ into disjoint clusters of expected diameter $2/\mu$ can be obtained in expected $\mathcal{O}(\text{sort}(|E|) + \text{scan}(|E|)/\mu)$ I/Os. The expected number of clusters is at most $1 + \mu|V|$.*

Proof. Choosing the masters and constructing set $L(0)$ takes $\mathcal{O}(\text{scan}(|V|)) = \mathcal{O}(\text{scan}(|E|))$ I/Os. By the proof of Lemma 6.8, every remaining vertex is expected to be "captured" by some master after $1/\mu$ iterations. Hence, the expected number of iterations performed by the procedure UNDIRECTEDDFS is $1/\mu$. Iteration i takes $\mathcal{O}(\text{sort}(|E_i|) + \text{scan}(|E|))$ I/Os, where E_i is the set of edges adjacent to the vertices in $L(i)$, if procedure UNDIRECTEDBFS is modified as follows: Instead of retrieving the adjacency list of every vertex in $L(i)$ using a random disk access, scan all adjacency lists and retrieve the contents of the adjacency lists of the vertices in $L(i)$. Since every edge of G appears in exactly two adjacency lists, the I/O-bound follows. The bound on the number of clusters is an immediate consequence of Observation 6.1. The bound on the expected diameter of each cluster is shown in Lemma 6.8. □

Breadth-first search. To construct a BFS-tree of G rooted at vertex r, the algorithm now applies procedure UNDIRECTEDBFS again. Before doing so, however, the representation of graph G is modified as follows: (1) The adjacency lists of all vertices in a cluster C_i are concatenated to form file $\mathcal{F}_i$. In particular, the edges in each file $\mathcal{F}_i$ are stored consecutively. (2) Every edge $(v, w) \in \mathcal{F}_i$ is represented as the triple (v, w, p_j), where $w \in C_j$ and p_j is the disk address of the first edge in $\mathcal{F}_j$.

In order to use this preprocessed representation of G effectively, procedure UNDIRECTEDBFS is modified as follows: The algorithm maintains a pool $\mathcal{H}$ that is guaranteed to contain all edges connecting vertices in the current level $L(i)$ with vertices in the next level $L(i+1)$ to be constructed; but $\mathcal{H}$ may also contain edges connecting vertices at levels greater than i. The edges in $\mathcal{H}$ are sorted by their source vertices. Also, as we will see, every level is produced in sorted order by the algorithm, so that in particular the current level $L(i)$ has been produced in sorted order by the previous iteration. The algorithm scans lists $L(i)$ and $\mathcal{H}$ to identify all vertices in $L(i)$ whose adjacency lists are not contained in $\mathcal{H}$. For each such vertex v, let C_j be the cluster containing vertex v. Then the address of file $\mathcal{F}_j$ is appended to a list Q. Once list Q has been produced, this list is sorted and duplicates are removed in a single scan. For every remaining entry in Q, the corresponding file $\mathcal{F}_j$ is appended to a list $\mathcal{H}'$. Then the edges in $\mathcal{H}'$ are sorted by their source vertices, and $\mathcal{H}$ and $\mathcal{H}'$ are merged. This ensures that pool $\mathcal{H}$ now contains the adjacency lists of all vertices in $L(i)$. Hence, a single scan of lists $L(i)$ and $\mathcal{H}$ suffices to extract these adjacency lists from $\mathcal{H}$ and create the list $X(i)$ of vertices adjacent to vertices in $L(i)$. Then list $X(i)$ is sorted and scanned to remove duplicates. Level $L(i+1)$ is now constructed from lists $L(i-1)$, $L(i)$ and $X(i)$ as before.

To analyze the I/O-complexity of this modified version of procedure UNDIRECTEDBFS, we split the cost into three parts: (1) I/Os spent on retrieving and sorting all files $\mathcal{F}_1, \dots, \mathcal{F}_q$. (2) I/Os spent on merging $\mathcal{H}$ and $\mathcal{H}'$. (3) I/Os spent on constructing list $L(i+1)$ from lists $L(i-1)$, $L(i)$ and $\mathcal{H}$.

The I/O-complexity for retrieving and sorting all files is $\mathcal{O}(|V'| + \text{sort}(|E|))$ because there are $|V'|$ files of total size $2|E|$. Since $E[|V'|] = \mu|V|$, the expected cost of retrieving and sorting all files is hence $\mathcal{O}(\mu|V| + \text{sort}(|E|))$. The cost of merging $\mathcal{H}$ and $\mathcal{H}'$ is $\mathcal{O}(\text{scan}(|\mathcal{H}|+|\mathcal{H}'|))$. Since every edge is contained in $\mathcal{H}'$ only once, the edges in $\mathcal{H}'$ contribute $\mathcal{O}(\text{scan}(|E|))$ to the total cost of this operation, summed over all iterations. To bound the total cost contributed by the edges in $\mathcal{H}$, we use Lemma 6.8. In particular, since the expected diameter of a cluster is $2/\mu$, we expect the algorithm to take at most $2/\mu$ iterations after discovering the first vertex in a cluster before all vertices in the cluster are discovered. Hence, once the corresponding file $\mathcal{F}_i$ has been incorporated into $\mathcal{H}$, all edges in $\mathcal{F}_i$ are expected to be removed from $\mathcal{H}$ after at most $2/\mu$ iterations. That is, the total size of $\mathcal{H}$, summed over all iterations, is expected to be $4|E|/\mu$, so that the expected cost of merging lists $\mathcal{H}$ and $\mathcal{H}'$ for all iterations is $\mathcal{O}(\text{scan}(|E|)/\mu)$.

The cost of computing list $L(i+1)$ from lists $L(i-1)$, $L(i)$ and $\mathcal{H}$ is $\mathcal{O}(\text{sort}(|E_i|) + \text{scan}(|L(i-1)| + |L(i)| + |\mathcal{H}|))$, where E_i is the set of edges incident to the vertices in $L(i)$. The total size of sets $E_0, E_1, \ldots$ is $2|E|$; the total size of sets $L(0), L(1), \ldots$ is $|V|$; and as argued above, the total size of list $\mathcal{H}$, summed over all iterations, is $\mathcal{O}(|E|/\mu)$. Hence, the cost for computing all levels is $\mathcal{O}(\mathrm{sort}(|E|) + \mathrm{scan}(|E|)/\mu)$, and we obtain the following lemma.

Lemma 6.10. *Given the preprocessing performed by the first phase of the algorithm, a BFS-tree of G can be computed in expected $\mathcal{O}(\mu|V| + \mathrm{sort}(|E|) + \mathrm{scan}(|E|)/\mu)$ I/Os.*

By Lemmas 6.9 and 6.10, the I/O-complexity of the improved BFS-algorithm is $\mathcal{O}(\mu|V| + \mathrm{sort}(|E|) + \mathrm{scan}(|E|)/\mu)$. By choosing $\mu = \min\bigl(1, \sqrt{|V|B/|E|}\bigr)$, we obtain the desired result.

Theorem 6.11. *A BFS-tree of an undirected graph $G = (V, E)$ can be computed in expected $\mathcal{O}\bigl(\sqrt{|V||E|/B} + \mathrm{sort}(|E|)\bigr)$ I/Os.*

A deterministic clustering algorithm. It is not clear how to make the randomized clustering algorithm of Section 6.3 achieve its I/O-complexity with high probability. Instead, the algorithm can be made deterministic rather easily. All that is required is a deterministic method for partitioning G into $\mathcal{O}(\mu|V|)$ disjoint clusters of diameter $\mathcal{O}(1/\mu)$. Such a partition can be obtained using an Euler tour of an arbitrary spanning tree T of G. More precisely, observe that an Euler tour of T has length $2|V| - 2$ and can hence be partitioned into $2\mu(|V|-1) = \mathcal{O}(\mu|V|)$ segments of length $1/\mu$. Each segment defines a cluster C_i. A vertex v of G may be in more than one segment. Then w.l.o.g. v is chosen to be in the cluster C_i with smallest index that corresponds to a segment containing v.

The crucial observation is that the BFS-phase of the randomized algorithm does not require the clusters to be connected. The only property that is used is that the expected distance in G between any two vertices in the same cluster is $\mathcal{O}(1/\mu)$. Since two vertices in the same cluster formed by the above deterministic procedure have distance at most $1/\mu$ from each other, it is now *guaranteed* that once a vertex in a cluster is discovered, all vertices in the cluster are discovered within the next $1/\mu$ iterations.

By Lemma 5.7 and the remark on Page 19, a spanning tree of G can be computed in $\mathcal{O}(\mathrm{sort}(|E|) \log_2 \log_2(|V|B/|E|))$ I/Os. Given a spanning tree of G, the Euler tour can be computed in $\mathcal{O}(\mathrm{sort}(|V|)) = \mathcal{O}(\mathrm{sort}(|E|))$ I/Os. The computation of clusters from the Euler tour requires a constant number of sorts and scans. Hence, the clustering phase of the algorithm takes $\mathcal{O}\bigl(\sqrt{|V||E|/B} + \mathrm{sort}(|E|) \log_2 \log_2(|V|B/|E|)\bigr)$ I/Os, as does the BFS-phase.

Theorem 6.12. *A BFS-tree of an undirected graph $G = (V, E)$ can be computed in $\mathcal{O}\bigl(\sqrt{|V||E|/B} + \mathrm{sort}(|E|) \log_2 \log_2(|V|B/|E|)\bigr)$ I/Os.*

7 Single Source Shortest Paths

In this section we discuss an algorithm for the single source shortest path problem on undirected graphs due to Kumar and Schwabe [22]. The algorithm is an

I/O-efficient version of Dijkstra's algorithm. In order for Dijkstra's algorithm to be I/O-efficient, an I/O-efficient priority queue and an I/O-efficient method for testing for previously visited vertices are needed. Ideally it would also be desirable to have an I/O-efficient data structure for retrieving the adjacency lists of the vertices in the graph. For BFS we have seen in the previous section that a clustering approach can be applied to at least reduce the number of random accesses performed while retrieving adjacency lists. For the single source shortest path problem this approach does not seem to work because there is no guarantee any more how long an adjacency list would remain in the pool $\mathcal{H}$ before it is removed. More importantly, a shortest path tree cannot easily be built level by level, as algorithm UNDIRECTEDBFS does for a BFS-tree. Hence, we shall be content with spending one I/O per vertex, as long as the number of I/Os spent per edge can be kept small.

In Section 7.1 we discuss the I/O-efficient priority queue used in the algorithm. In Section 7.2 we discuss the shortest path algorithm and show how it makes use of a second priority queue to avoid having to check for visited vertices.

7.1 The Tournament Tree

As discussed in the chapter by Lars Arge, the buffer tree [2] can be used as a priority queue that can process a sequence of N INSERT, DELETE, and DELETEMIN operations in $\mathcal{O}(\text{sort}(N))$ I/Os. Unfortunately this priority queue does not support a DECREASEKEY operation, which is required by Dijkstra's algorithm, unless this operation can be simulated by a DELETE operation followed by an insertion. The latter is possible only if the previous priority of the element is known, which in general is hard to achieve in Dijkstra's algorithm.

In this section we discuss the external tournament tree proposed by Kumar and Schwabe [22], which supports INSERT, DELETE, DELETEMIN and DECREASEKEY operations at an amortized cost of $\mathcal{O}\big(\frac{1}{B}\log_2\frac{N}{B}\big)$ I/Os per operation and uses $\mathcal{O}(N/B)$ blocks of external memory. In these bounds N denotes the total number of elements that may potentially be stored in the priority queue. In particular, N may be much larger than the actual number of elements stored in the priority queue, which could affect the efficiency of the data structure. However, in most graph algorithms, $N = \mathcal{O}(|V|)$ or $N = \mathcal{O}(|E|)$, so that the tournament tree pays only a $\log_2(M/B)$ factor in performance compared to the buffer tree, for the added benefit of supporting the DECREASEKEY operation.

The data structure. So let X be the set of elements potentially stored in the priority queue, and assume that the elements in X are numbered 1 through $N = |X|$. The numbering is required to establish a total order on the elements of X and to compare two elements quickly w.r.t. this total order. For the sake of simplifying the description of the data structure, we also assume that N is a multiple of M.

The *tournament tree* is a *static* binary tree T with the following properties:

(i) Tree T has N/M leaves.

(ii) All leaves of T are at level $d = \lfloor \log_2(N/M) \rfloor$ or $d-1$.

(iii) Let the leaves of T be numbered from left to right. Then the elements of X numbered $(i-1)M+1$ through iM map to the i-th leaf of T. An element x of X is stored either at the leaf $l(x)$ it maps to or at an ancestor thereof.

(iv) A node stores between $M/2$ and M elements. The priorities of the elements stored at any node are smaller than the priorities of the elements stored at its descendants.

(v) Each internal node has an associated signal buffer of size M. This buffer stores update signals that are used to propagate updates of T down the tree towards the leaves.

(vi) The root of T is held in main memory.

Since T stores all elements of X at all times, we need a criterion to decide when an element stored in T is not in the subset of X currently represented by T. The adopted convention is that an element is in this subset if and only if its priority is finite. Hence, by initializing all elements in T to have infinite priority, tree T initially represents the empty set.

Priority queue operations. Given a tournament tree T, the only operation that requires immediate processing is the DELETEMIN operation. By Property (iv), the element with minimum priority in T is stored at the root. By Property (vi), the root of T is held in main memory. Hence, a DELETEMIN operation can be performed without incurring any I/Os by extracting the element with minimum priority stored at the root of T. To maintain the invariant that an element that is "not stored" in T is stored in T with infinite priority, the retrieved element has to be inserted into T with priority ∞. This is achieved by sending signal UPDATE(x, ∞) to the root of T (see the discussion on signals below).

Operations INSERT, DELETE, and DECREASEKEY are realized using signals that are sent to the root and then propagate down the tree towards the leaves. When a signal reaches a node $v \in T$, it is first *applied* to v — that is, it effects certain changes to the set of elements stored at v — and then the signal itself or another, newly generated, signal is sent to one or both of the children of v.

To perform a DELETE operation, a DELETE signal is sent to the root of T. Operations INSERT and DECREASEKEY are both realized using an UPDATE signal. Next we describe the effects of sending these signals to a node $v \in T$.

DELETE(x): If element x is stored at node v, it is deleted, and signal UPDATE(x, ∞) is sent to the next node w on the path to leaf $l(x)$. If x is not stored at v, signal DELETE(x) is sent to w.

UPDATE(x, p): If element x is stored at v, its priority is updated to $\min(p, p')$, where p' is its current priority. If x is not stored at v, and all elements stored at v have priority less than p, signal UPDATE(x, p) is propagated to the next node w on the path to leaf $l(x)$. Finally, if there is an element with priority $p' \geq p$ stored at v, element x is added to the set of elements stored at v. After this update, any other copy of x with finite priority $p'' \geq p$ possibly stored at a descendant of v has to be removed from T. This is achieved by sending signal DELETE(x) to w.

The insertion of element x into the set of elements stored at node v may cause this set to overflow because it already contains M elements. If this happens, the element z with maximal priority p_z in this set is moved to the child of v on the path to leaf $l(z)$ by sending signal PUSH(z, p_z) to this child. Finally, a signal UPDATE(x, ∞) is handled in a special way when it reaches leaf $l(x)$. When this happens, the signal makes sure that element x is stored with priority ∞ at this leaf by inserting element x if necessary.

PUSH(x, p): This signal inserts element x into the set of elements stored at node v. If this set already contains M elements, the element z with maximum priority p_z in this set is moved to the child of v on the path to leaf $l(z)$ by sending signal PUSH(z, p_z) to this child.

We leave it as an exercise to verify that the implementation of all priority queue operations using the above signals updates the set of elements stored in T and their priorities correctly. Moreover, Properties (i)-(vi) are maintained, except for the possible underflow of a node after applying a DELETEMIN or DELETE operation to it. We show how to deal with these underflows below.

Lazy signal propagation. So far we have assumed that all signals are sent all the way down to the leaves when generated by an update operation. However, this is not necessary, as long as the root of T always stores the elements with smallest priority in the set tree T is supposed to represent. Hence, after applying an update signal to the root, the sending of signals to its children can be delayed until either enough of them have been collected to guarantee that they can be applied I/O-efficiently to the children of the root or an underflow of the root requires to move elements from the children of the root to the root. When applying signals to any node $v \in T$, the same strategy can be applied to delay the sending of signals to its children. Intuitively, whenever a subtree of T is affected by an update, it suffices to update its root and delay updates of its descendants until the updates of the root cannot be performed without fetching data from its children.

This delayed propagation of signals down the tree is realized using the signal buffers of the nodes in T. After a signal has been applied to a node $v \in T$, the signals to be sent to v's children of v are appended to v's signal buffer instead of sending them to v's children immediately. As soon as v's signal buffer contains at least M elements, it is emptied. This operation is performed as follows: Scan the set S of signals in the buffer and partition them into two sets S_u and S_w for the two children, u and w, of v. Load the set of elements stored at node u into main memory, scan set S_u, and apply the signals in S_u to this set of elements. Append the signals generated during this update of node u to u's signal buffer. Now repeat the whole procedure to apply the signals in S_w to w. As a result of these updates, the signal buffers of nodes u and w may overflow. If this happens, these buffers are emptied recursively.

Excluding the recursive emptying of the signal buffers of v's children, emptying the buffer of node v takes $\mathcal{O}(\text{scan}(|S| + M)) = \mathcal{O}(\text{scan}(|S|))$ I/Os because nodes u and w store $\mathcal{O}(M)$ elements and $|S| \geq M$. Hence, every signal involved in a buffer-emptying process costs $\mathcal{O}(1/B)$ I/Os amortized. Since every signal is involved in at most $\mathcal{O}(\log_2(N/B))$ buffer-emptying processes, one per level, the amortized cost per signal is $\mathcal{O}\big(\frac{1}{B}\log_2\frac{N}{B}\big)$. Next we argue that every priority queue operation generates $\mathcal{O}(1)$ signals, so that the cost for propagating signals down the tree is $\mathcal{O}\big(\frac{1}{B}\log_2\frac{N}{B}\big)$ amortized per priority queue operation.

Every priority queue operation sends one signal to the root of T. A Delete signal propagates down the tree until it finds the element to be deleted, at which point it is replaced by an Update signal. An Update signal travels down the tree until it either terminates following the update of the priority of the targeted element, or it is replaced by a Delete and possibly a Push signal when it causes the insertion of the targeted element into the set stored at some node of T. A Push signal propagates down the tree, possibly changing the element it "carries", until it finds a node where there is room to insert the current element. Hence, the only signal that can possibly split into two signals on its way down the tree is an Update signal. The generated Push signal does not multiply. We have to argue that the generated Delete signal does not multiply either. To do this, we show that the Update signal generated by a Delete stays an Update signal, i.e., does not split. To see that this is true, observe that an Update signal is replaced by a Delete and a Push signal only if it encounters a node that stores an element with higher priority than its own; but this is impossible for an Update signal generated by a Delete signal because its priority is ∞. Hence, every priority queue operation sends at most two signals down the tree.

Filling underfull nodes. So far we have conveniently ignored what happens when a node v stores less than $M/2$ elements as the result of a DeleteMin or Delete operation. When this happens, elements stored at v's children have to be moved to v. This in turn may cause v's children to underflow, so that they have to be filled with elements from their children, and so on. Hence, even a DeleteMin operation, which otherwise does not incur any I/Os, may cause a considerable number of I/Os to be performed. However, we show next that the amortized cost for filling underfull nodes in this manner is only $\mathcal{O}\big(\frac{1}{B}\log_2\frac{N}{B}\big)$ per operation.

What precisely happens when a node v underflows is that the $M/2$ elements with smallest priority stored at v's children are moved to v. To guarantee that the sets of elements stored at v's children are up-to-date, v's signal buffer has to be emptied before moving elements from v's children to v. The emptying of v's signal buffer costs $\mathcal{O}(\text{scan}(M))$ I/Os, which follows from the discussion of the buffer-emptying process above and the fact that v's signal buffer contains at most M elements because otherwise it would have been emptied already. To move the $M/2$ elements with smallest priority from v's children to v, it suffices to scan the two sets stored at v's children, which takes $\mathcal{O}(\text{scan}(M))$ I/Os. Hence, the total cost of filling v's signal buffer with $M/2$ elements from its children is

$\mathcal{O}(\text{scan}(M))$, $\mathcal{O}(1/B)$ I/Os amortized per element. The moving of elements from v's children to v may leave v's children underfull, so that they have to be filled recursively. However, the I/Os required to do this can be charged to the elements that are moved to v's children. We observe that every level an element travels up the tree costs $\mathcal{O}(1/B)$ I/Os amortized.

Since the tournament tree is initially empty, elements that move up the tree first have to be moved down the tree by means of signals. For every level an element travels up the tree, we can hence charge the signal that moved the element in the opposite direction. This increases the amortized cost per signal by only a constant factor and hence changes the amortized cost per priority queue operation by only a constant factor. Thus, we obtain the following theorem.

Theorem 7.1. *Using an I/O-efficient tournament tree, a sequence of K* INSERT, DELETE, DELETEMIN, *and* DECREASEKEY *operations can be processed in $\mathcal{O}\left(\frac{K}{B}\log_2\frac{N}{B}\right)$ I/Os.*

7.2 An I/O-Efficient Version of Dijkstra's Algorithm

Dijkstra's algorithm [14] can be made I/O-efficient using the tournament tree as the priority queue that stores the vertices of graph $G = (V, E)$ sorted according to their tentative distances from the source s. However, replacing the internal memory priority queue of choice with the tournament tree is not sufficient to immediately obtain an I/O-efficient shortest path algorithm. The problem is that Dijkstra's algorithm tests every neighbor w of the current vertex v whether it has already been finished[5] before trying to update its tentative distance using a DECREASEKEY operation. If there is no way to avoid these tests, the algorithm spends one I/O per edge of G, $\mathcal{O}(|E|)$ I/Os in total.

To avoid performing these tests, the shortest path algorithm of [22] performs an UPDATE operation for *all* neighbors of v, excluding its parent in the shortest path tree, irrespective of whether or not they are finished. While this avoids the expensive test for finished vertices, it creates the following problem: Let u be a neighbor of v that has already been finished, and let $\{u, v\}$ be the edge connecting u and v in G. Then the algorithm re-inserts u into priority queue Q with priority $\text{dist}(s, v) + \omega(\{u, v\})$, where $\omega(e)$ denotes the weight of edge e. This will ultimately cause u to be visited for a second time, which is incorrect. We call such a re-insertion of u a *spurious update*. Next we discuss a method that guarantees that the copy of u inserted by a spurious update is deleted from Q using a DELETE operation before it can cause a second visit to vertex u.

The method to achieve this is based on the observation that a neighbor u of v that is finished before v performs an update of v before v is finished. By recording this update attempt of u on v in a second priority queue Q', this information can later be used to prevent the spurious update of v on u from doing any harm. In particular, when vertex u attempts to update v's tentative

[5] A vertex v is *finished* when the algorithm has determined the final distance of v from s and has inserted v's neighbors into the priority queue.

distance, vertex u is inserted into Q' with priority $\text{dist}(s, u) + \omega(\{u, v\})$. The next vertex to be visited by the algorithm is now determined from the outcome of two DELETEMIN operations, one on Q and one on Q'.

Let (v, p_v) be the entry retrieved from Q, and let (w, p_w) be the entry retrieved from Q'. If $p_w < p_v$, entry (v, p_v) is re-inserted into Q, vertex w is deleted from Q by applying a DELETE(w) operation to Q, and then the whole procedure is iterated. If $p_v \leq p_w$, entry (w, p_w) is re-inserted into Q', and vertex v is visited as normal. Let us show that this method achieves the desired goal.

Lemma 7.2. *A spurious update is deleted before the targeted entry can be retrieved using a* DELETEMIN *operation.*

Proof. Consider a vertex v and a neighbor u of v that is finished before v, so that v performs a spurious update on u. Denote the spurious update as event A, the deletion of the re-inserted copy of u as event B, and the extraction of the re-inserted copy of u using a DELETEMIN operation as event C. We have to show that event B happens after event A, but before event C can occur.

Assume that all vertices have different distances from s.[6] Under this assumption $\text{dist}(s, u) < \text{dist}(s, v)$ because u is finished before v. Moreover, $\text{dist}(s, v) \leq \text{dist}(s, u) + \omega(\{u, v\})$. The latter implies that event B happens after event A because event A happens when vertex v is retrieved from Q with priority $\text{dist}(s, v)$, and event B happens when the copy of u inserted into Q' with priority $\text{dist}(s, u) + \omega(\{u, v\})$ is retrieved from Q'. The former implies that $\text{dist}(s, u) + \omega(\{u, v\}) < \text{dist}(s, v) + \omega(\{u, v\})$, so that event B happens before event C. This proves the lemma. □

Lemma 7.2 shows that the modified version of Dijkstra's algorithm described above is correct. It remains to analyze its I/O-complexity. The algorithm spends $\mathcal{O}(|V| + \text{scan}(|E|))$ I/Os to access all adjacency lists because every adjacency list is touched once, namely when the corresponding vertex is finished. The number of priority queue operations performed by the algorithm is $\mathcal{O}(|E|)$: Every edge of G causes two insertions into priority queue Q' and two updates of priority queue Q, one each per endpoint. All other priority queue operations can be partitioned into sequences of constant length so that each sequence decreases the total number of elements stored in Q and Q' by at least one. Hence, only $\mathcal{O}(|E|)$ such sequences are executed. Using a tournament tree as priority queue Q and a buffer tree [2] as priority queue Q', the total cost of all priority queue operations is hence $\mathcal{O}\left(\frac{|E|}{B} \log_2 \frac{|E|}{B}\right)$, and we obtain the following result.

Theorem 7.3. *The single source shortest path problem on an undirected graph $G = (V, E)$ can be solved in $\mathcal{O}\left(|V| + \frac{|E|}{B} \log_2 \frac{|E|}{B}\right)$ I/Os.*

Remark. In the proof of Lemma 7.2 we assume that no two vertices have the same distance from s. It is not hard to see that the proof remains correct if no

[6] If this is not the case, the algorithm needs to be modified. See the remark at the end of this section.

two vertices with the same distance are adjacent. In order to handle adjacent vertices with the same distance, the algorithm has to be modified. In particular, all vertices with the same distance have to be processed simultaneously, similar to the simultaneous construction of levels in the BFS-algorithm from Section 6.2. The reason for this is that there seems to be no way to guarantee that for two adjacent vertices v and w at the same distance from s, non-spurious updates and deletions of spurious updates are processed in the correct order. By processing all vertices at the same distance from s at the same time, it can be guaranteed that these vertices do not update each other's distances at all. The problem with adjacent vertices that have the same distance from s has been noticed in [22]; but the proposed solution is incorrect.

8 Shortest Paths in Planar Graphs

Given that all algorithms for graph searching problems such as BFS, DFS and SSSP spend considerably more I/Os than the lower bound if the graph is sparse, a number of researchers [4, 5, 21, 24–26, 32] have tried to exploit the structure of special classes of sparse graphs in order to solve these problems I/O-efficiently on graphs in these classes. In the remainder of this course we focus on planar graphs and discuss how to solve the above three problems in $\mathcal{O}(\text{sort}(N))$ I/Os. (From now on we use N to denote the size of the vertex set of the given graph G.) For the sake of simplicity we assume that an embedding of the graph is provided as part of the input. This is not a serious restriction because such an embedding can be obtained in $\mathcal{O}(\text{sort}(N))$ I/Os [26, 32].

First we focus on shortest paths and BFS. More precisely, we discuss a shortest path algorithm by Arge *et al.* [4], which of course can also be used to compute a BFS-tree of a planar graph. We assume that the given graph G has degree three[7] and that a regular B^2-partition of G is given. Such a partition is defined as follows: Given a planar graph $G = (V, E)$, a *regular h-partition* of G is a pair $\mathcal{P} = (S, \{G_1, \dots, G_k\})$, where S is a subset of the vertices of G and graphs $G_1, \dots, G_k$ are disjoint subgraphs of $G - S$ with the following properties:

(i) $G_1 \cup \dots \cup G_k = G - S$.
(ii) For every edge in $G - S$, the two endpoints are in the same graph G_i. (That is, each graph G_i is the union of a number of connected components of $G - S$.)
(iii) $|S| = \mathcal{O}(N/\sqrt{h})$.
(iv) $k = \mathcal{O}(N/h)$.
(v) Every graph G_i has at most h vertices.
(vi) Every graph G_i is adjacent to at most $\sqrt{h}$ vertices in S. This subset of S is called the *boundary* ∂G_i of G_i.
(vii) Let $S_1, \dots, S_t$ be a partition of S into subsets so that the vertices in each subset are adjacent to the same set of subgraphs G_i. Then $t = \mathcal{O}(N/h)$. Sets $S_1, \dots, S_t$ are called the *boundary sets* of partition $\mathcal{P}$. (See Figure 8.1.)

[7] The degree of a graph is the maximum degree of its vertices.

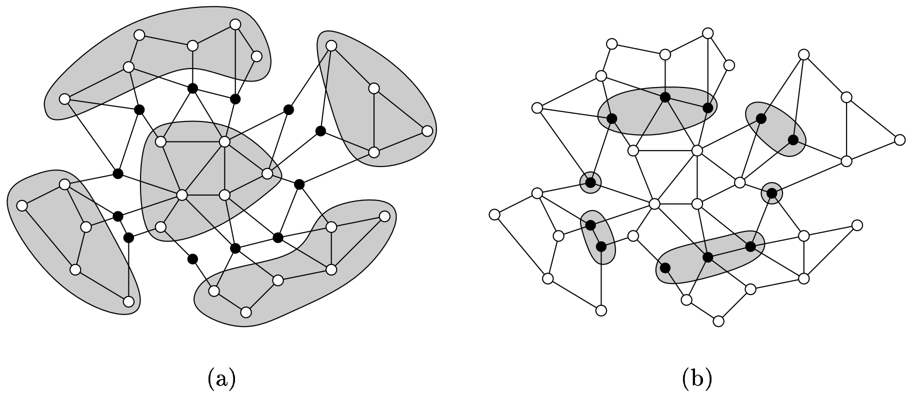

Figure 8.1
(a) A partition of a planar graph into the shaded subgraphs using the black separator vertices. (b) The boundary sets of this partition.

The vertex set S is also referred to as the *separator* that induces partition $\mathcal{P}$. We discuss in Section 9 how to obtain a partition satisfying Properties (i), (ii), (iii) and (v). The other properties can be ensured using fairly simple modifications of the algorithm discussed in Section 9. For details the reader may refer to [32].

One additional assumption we make is that the amount of available main memory is large enough to hold a planar graph with $B^2 + B + 1$ vertices.

Outline. The algorithm of [4] solves the SSSP problem in three steps (see Algorithm 8.1). The first step replaces each subgraph $\tilde{G}_i$ of G induced by the vertices in $V(G_i) \cup \partial G_i$ with a complete graph G'_i over the vertices in ∂G_i (see Figure 8.2). Graph G'_i has the property that for any two vertices $v, w \in \partial G_i$, their distances from each other in $\tilde{G}_i$ and G'_i are the same. As we show below, this implies the property stated as a comment of Step 2 of the algorithm, namely that the distances from s to all separator vertices are preserved in the resulting graph G_R.[8] Hence, their distances from s in G can be computed by solving the single source shortest path problem on G_R, as done in the second step of the algorithm. Finally, in the third step, the algorithm exploits the fact that for any vertex v in G_i, the shortest path from s to v in G consists of a shortest path from s to a vertex x in ∂G_i followed by a shortest path from x to v in $\tilde{G}_i$.

Correctness. The following two lemmas formally prove the two structural properties used by the algorithm and establish its correctness.

Lemma 8.1. *For any vertex $v \in S$, $dist_G(s, v) = dist_{G_R}(s, v)$.*

[8] For this to be true, s has to be in G_R, which is true only if $s \in S$. The latter can easily be enforced.

```
Procedure PlanarSSSP
1: Construct a compressed graph $G_R$ that captures the distance between separator
   vertices:
     $G_R \leftarrow (S, \emptyset)$
     for every graph $G_i$ in partition $\mathcal{P}$ do
       Let $\tilde{G}_i$ be the subgraph of $G$ induced by all vertices in $V(G_i) \cup \partial G_i$.
       Compute the distance in $\tilde{G}_i$ from every vertex in $\partial G_i$ to every other vertex
       in $\partial G_i$.
       Add an edge $\{v, w\}$ to $G_R$ for every pair $v, w$ of vertices in $\partial G_i$. The weight
       of edge $\{v, w\}$ is the distance from $v$ to $w$ in $\tilde{G}_i$.
     end for
2: Compute the distances from $s$ to all separator vertices in $G_R$.
     {Every separator vertex $v$ has the same distance from $s$ in $G$ and $G_R$.}
3: Compute the distances from $s$ to all vertices in $G$:
     for every graph $G_i$ in partition $\mathcal{P}$ do
       Let $\tilde{G}_i$ be the subgraph of $G$ induced by all vertices in $V(G_i) \cup \partial G_i$.
       Add vertex $s$ to $\tilde{G}_i$ and connect $s$ to every vertex in $\partial G_i$. The weight of edge
       $\{s, v\}$, $v \in \partial G_i$, is the distance from $s$ to $v$ computed in Step 2. Let the
       resulting graph be $G''_i$.
       Compute the distance from $s$ to all vertices in $G''_i$.
     end for
```

Algorithm 8.1
A shortest path algorithm for planar graphs.

Proof. Consider any path $P = (s = x_0, x_1, \ldots, x_k = v)$ from s to v in G, and let $0 = i_1 < i_2 < \cdots < i_q = k$ be the indices so that vertices $x_{i_1}, \ldots, x_{i_q}$ are separator vertices on this path. Then every subpath $(x_{i_j}, \ldots, x_{i_{j+1}})$ stays completely inside some graph $\tilde{G}_i$. Since the weight of edge $\{x_{i_j}, x_{i_{j+1}}\}$ in G_R equals the length of the shortest path from x_{i_j} to $x_{i_{j+1}}$ in $\tilde{G}_i$, replacing path $(x_{i_j}, \ldots, x_{i_{j+1}})$ in P with edge $\{x_{i_j}, x_{i_{j+1}}\}$ results in a path P' whose length is at most that of P. By doing this for all subpaths of P connecting two consecutive separator vertices, we obtain a path P_R in G_R whose length is at most that of P. Conversely, given a path P_R from s to v in G_R, every edge $\{v, w\}$ in P_R represents a path from v to w in G. Hence, replacing each edge in P_R by the corresponding path in G, we obtain a path P in G whose length is the same as that of P_R. □

Lemma 8.2. *For any graph G_i, $1 \leq i \leq k$, in partition $\mathcal{P}$ and any vertex $v \in G_i$, $dist_G(s, v) = dist_{G'_i}(s, v)$.*

Proof. Consider any path P from s to some vertex v in G_i, and let x be the last separator vertex on this path. Assume that $s \neq x$. (If $s = x$, the proof becomes simpler.) Let P_1 be the subpath of P from s to x, and let P_2 be the subpath of P from x to v. By Lemma 8.1, there exists a path from s to x in G_R whose length is at most that of P_1, so that edge $\{s, x\} \in G'_i$ has length at most that of path P_1.

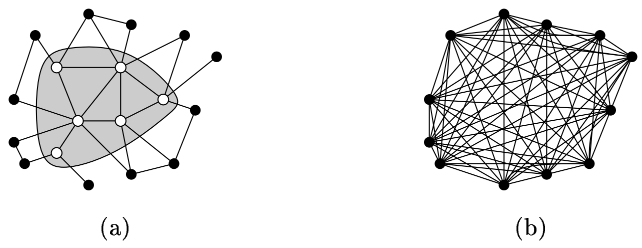

Figure 8.2
(a) The central graph G_i in the partition of Figure 8.1a and its boundary vertices. (b) The corresponding graph G'_i.

Path P_2 exists also in G'_i. Hence, by concatenating edge $\{s, x\}$ with path P_2, we obtain a path of length at most that of P from s to v in G'_i. Conversely, given a path P' from s to v in G'_i, the first edge $\{s, x\}$ on the path can be replaced by a path of the same length in G_R, which in turn can be replaced by a path P_1 of the same length in G, by Lemma 8.1. Hence, the concatenation of P_1 with the subpath P'_2 of P' from x to v produces a path from s to v in G whose length is the same as that of P'. □

Complexity. Given that the main memory is large enough to hold a planar graph with $B^2 + B + 1$ vertices, Steps 1 and 3 take $\mathcal{O}(\text{sort}(N))$ I/Os because the required shortest path computations can be carried out in main memory.

To execute Step 1 of the algorithm, the first thing that needs to be done is compute graphs $\tilde{G}_1, \dots, \tilde{G}_k$, i.e., store their vertex and edge sets consecutively on disk. The vertex set of graph $\tilde{G}_i$ is the set of endpoints of all edges that have at least one endpoint in G_i. The edge set of $\tilde{G}_i$ contains all those edges of G that have both endpoints in $V(\tilde{G}_i)$. Assuming that partition $\mathcal{P}$ is represented by an appropriate labelling of the vertices of G, it suffices to sort and scan the vertex and edge sets of G a constant number of times to extract graphs $\tilde{G}_1, \dots, \tilde{G}_k$. We have seen this type of computation in previous sections and omit the details.

Once graphs $\tilde{G}_1, \dots, \tilde{G}_k$ have been identified, they can now be loaded into main memory, one at a time, the shortest path computation of Step 1 can be carried out in main memory because each graph $\tilde{G}_i$ has at most $B^2 + B$ vertices, and the edges of G_R can be written to disk in a linear number of I/Os. Hence, Step 1 takes $\mathcal{O}(\text{sort}(N))$ I/Os.

To execute Step 3, the distances of all vertices in S from the source s have to be copied from their copies in G_R to their copies in graphs $\tilde{G}_1, \dots, \tilde{G}_k$. This can again be done in $\mathcal{O}(\text{sort}(N))$ I/Os. After that, each graph $\tilde{G}_i$, $1 \leq i \leq k$, is loaded into main memory for a second time, and the shortest path computation of Step 3 can be performed without incurring any further I/Os.

In the remainder of this section we discuss a method to solve the SSSP problem on graph G_R in $\mathcal{O}(\text{sort}(N))$ I/Os, so that the whole algorithm takes $\mathcal{O}(\text{sort}(N))$ I/Os.

Shortest paths in G_R. One can come close to solving the SSSP problem on graph G_R in $\mathcal{O}(\text{sort}(N))$ I/Os by observing that this graph has $\mathcal{O}(N/B)$ vertices and $\mathcal{O}(N)$ edges. Indeed, its vertex set is S, and every edge in G_R belongs to some graph G'_i. There are $\mathcal{O}(N/B^2)$ such graphs $G'_1, \dots, G'_k$, and each of them has at most B^2 edges. From this observation it follows that the SSSP problem on G_R can be solved in $\mathcal{O}\left(\frac{N}{B} \log_2 \frac{N}{B}\right)$ I/Os using the shortest path algorithm from Section 7.

The main obstacle preventing the improvement of this bound to $\mathcal{O}(\text{sort}(N))$ is that Dijkstra's algorithm requires a priority queue that supports a DECREASE-KEY operation; but no priority queue is known that supports this operation and processes a sequence of N updates in $\mathcal{O}(\text{sort}(N))$ I/Os. On the other hand, there are priority queues that support INSERT, DELETE, and DELETEMIN operations and process a sequence of N updates in $\mathcal{O}(\text{sort}(N))$ I/Os [2, 10]. The DELETE operation of these priority queues takes the element to be deleted and its current priority as an argument. That is, the priority of an element has to be known in order to delete it.

Arge *et al.* present a modified version of Dijkstra's algorithm that avoids the use of DECREASEKEY operations by exploiting the fact that graph G_R is derived from a regular B^2-partition of a planar graph of bounded degree. The algorithm maintains a list L storing the tentative distance of every vertex from s as well as a priority queue Q that stores the unfinished vertices of G. For every vertex in Q, its priority is the same as its tentative distance in L. Initially, all vertices in G_R, except s, have tentative distance (and priority) ∞.

In each step, the next vertex v to be finished is retrieved from Q using a DELETEMIN operation. Then the adjacency list of v is loaded into main memory, and for each vertex in the adjacency list, its tentative distance is retrieved from L. For every neighbor w of v so that the sum d' of $\text{dist}(s, v)$ and the weight of edge $\{v, w\}$ is less than the current distance d from s to w, its distance in L is changed to d'. Its priority in Q is decreased to d' by first deleting the current copy of w with priority d from Q and then inserting a new copy with priority d' into Q. That is, the required DECREASEKEY operation is simulated using a DELETE and an INSERT operation, which is possible because w's old priority d is known when performing the update. The algorithm repeats this procedure until all vertices of G are finished.

The I/O-complexity of this procedure can be split into the costs of retrieving the adjacency lists of all vertices, performing priority queue operations and accesses to list L. Retrieving the adjacency lists takes $\mathcal{O}(\text{scan}(N))$ I/Os because there are only $\mathcal{O}(N/B)$ vertices in G and the total size of all adjacency lists is $\mathcal{O}(N)$. The algorithm performs $\mathcal{O}(N)$ priority queue operations, two per edge, which takes $\mathcal{O}(\text{sort}(N))$ I/Os using a buffer tree [2] as priority queue. Finally, observe that list L is accessed $\mathcal{O}(N)$ times, $\mathcal{O}(1)$ times per edge. If the entries in L are not arranged carefully, the algorithm may spend one I/O per access, so that the procedure takes $\mathcal{O}(N + \text{sort}(N))$ I/Os. By arranging the vertices in L in a carefully chosen order, the number of I/Os spent on accessing list L can be reduced to $\mathcal{O}(N/B)$, which reduces the I/O-complexity of the algorithm to $\mathcal{O}(\text{sort}(N))$.

The order chosen for the vertices in L is so that the vertices in each boundary set of partition $\mathcal{P}$ are stored consecutively. The vertices in each boundary set S_j are on the boundary of the same subgraphs of G in partition $\mathcal{P}$ and hence have the same neighbors in G_R. That is, if one vertex in S_j needs to be retrieved from list L because one of its neighbors is finished, all other vertices in S_j also need to be retrieved from L. Instead of spending one I/O per access, these vertices can now be loaded in a blockwise fashion. More precisely, every boundary set S_j can be retrieved from L in $\mathcal{O}(1)$ I/Os because it is a subset of the boundary of some subgraph G_i in the partition and hence has size at most B. Since every vertex in G has degree at most three, it is on the boundary of at most three regions in G, so that every boundary set S_j is on the boundary of at most three regions. This implies that every vertex $v \in S_j$ has degree $\mathcal{O}(B)$ in G_R because the neighbors of v in G_R are the boundary vertices of these regions. We have argued above that boundary set S_j is accessed once for each such neighbor and that each access costs $\mathcal{O}(1)$ I/Os. Hence, the algorithm spends $\mathcal{O}(B)$ I/Os on accesses to boundary set S_j. Since there are $\mathcal{O}\left(N/B^2\right)$ boundary sets, the total number of I/Os spent on accessing list L is hence $\mathcal{O}\left(B \cdot N/B^2\right) = \mathcal{O}(N/B)$.

Procedure PlanarSSSP makes the assumption that $M = \Omega\left(B^2\right)$. As we will see in the next section, the best known algorithm to obtain a regular B^2-partition of a planar graph requires that $M = \Omega\left(B^2 \log^2 B\right)$, so that we obtain the following result.

Theorem 8.3. *Provided that $M = \Omega(B^2 \log^2 B)$, the single source shortest path problem on planar graphs with non-negative edge weights can be solved in $\mathcal{O}(\text{sort}(N))$ I/Os.*

Remark. Similar to the BFS-algorithms in Section 6, the SSSP-algorithm discussed above only computes the distance of every vertex from s. We leave it as an exercise to verify that once these distances are given, an $\mathcal{O}(\text{sort}(N))$ I/O postprocessing step is sufficient to extract a shortest path tree of G.

9 Planar Graph Partitions

Partitions of planar graphs using small separators are utilized in algorithms for problems such as solving sparse systems of linear equations, approximating solutions to NP-hard problems on planar graphs and, as we have seen, shortest paths in planar graphs. The main difficulty with computing a good partition of a planar graph I/O-efficiently is that all existing internal memory algorithms for this problem use BFS to partition the graph into levels and then judiciously use this partition to compute a small set of vertices whose removal partitions the graph into small subgraphs. Since the shortest path algorithm from Section 8 is the only known algorithm that computes a BFS-tree of a planar graph in $\mathcal{O}(\text{sort}(N))$ I/Os, and it requires a separator of the graph to be given as part of the input, this leads to circular dependencies between BFS and the problem of computing planar separators. In this section we discuss a separator algorithm by Maheshwari and Zeh [26] that applies graph contraction in a non-trivial way to obtain the desired partition without using BFS.

At the core of the algorithm is a graph hierarchy $G = G_0, G_1, \ldots, G_r$ whose properties guarantee that computing a partition of G_r using an internal memory algorithm does not cost too many I/Os and that a sufficiently good partition of each graph G_i can be derived I/O-efficiently from a partition of G_{i+1}. The main difficulty of the algorithm is computing this graph hierarchy.

In Section 9.1 we discuss the properties of graphs $G = G_0, G_1, \ldots, G_r$ and show how to exploit them to obtain an optimal partition of G in $\mathcal{O}(\text{sort}(N))$ I/Os. In Section 9.2 we discuss how this graph hierarchy can be computed in the same number of I/Os.

9.1 Computing the Partition

Let G be an embedded planar graph, let $h > 0$ be an integer so that the algorithm is asked to compute a set S of vertices whose removal partitions G into subgraphs of size at most h, and let $G = G_0, G_1, \ldots, G_r$ be a hierarchy of graphs with the following properties:

(i) $r = \log B$,
(ii) Graphs $G_0, \ldots, G_r$ are planar,
(iii) For $1 \leq i \leq r$, every vertex in G_i represents at most 56 vertices in G_{i-1},
(iv) For $0 \leq i \leq r$, every vertex in G_i represents at most 2^i vertices in G, and
(v) For $0 \leq i \leq r$, graph G_i has $\mathcal{O}(N/2^i)$ vertices.

Also assume that $M \geq 56h \log^2 B$. Then the desired partition of G can be obtained by computing a separator S_r of G_r and then deriving a separator S_i for each graph G_i, $0 \leq i < r$, from separator S_{i+1}. Each separator S_i has the property that it partitions graph G_i into subgraphs of size at most $h \log^2 B$. For graph G_r, separator $S_r = S''_r$ is computed using the linear-time internal memory algorithm of Aleksandrov and Djidjev [1]. Given separator S_{i+1}, the separator S_i for G_i is computed as follows: Let S'_i be the set of vertices in G_i represented by the vertices in S_{i+1}. Property (iii) of the graph hierarchy implies that no connected component of $G_i - S'_i$ has size exceeding $56h \log^2 B$. Since we assume that the main memory is large enough to hold a planar graph of this size, a partition of G_i into subgraphs of size at most $h \log^2 B$ can be obtained by loading each connected component of $G_i - S'_i$ into main memory and applying the algorithm of [1] again. Let S''_i be the set of separator vertices introduced by partitioning the connected components of $G_i - S'_i$ in this manner. Then separator S_i is the union of sets S'_i and S''_i.

The separator S_0 obtained in this manner partitions graph G into subgraphs of size at most $h \log^2 B$. The algorithm of [1] used to compute separators

$S_0'', \ldots, S_r''$ guarantees that $|S_i''| = \mathcal{O}\big(|G_i|/(\sqrt{h}\log B)\big)$. Hence, by Property (iv) of the graph hierarchy, the size of separator S_0 is

$$\begin{aligned}|S_0| &\leq \sum_{i=0}^{r} 2^i |S_i''| \\ &= \sum_{i=0}^{r} 2^i \mathcal{O}\left(|G_i| / \left(\sqrt{h}\log B\right)\right) \\ &= \sum_{i=0}^{r} 2^i \mathcal{O}\left(N / \left(2^i\sqrt{h}\log B\right)\right) \\ &= \mathcal{O}\left(N/\sqrt{h}\right).\end{aligned}$$

In order to obtain the final separator S, the connected components of $G - S_0$ are loaded into main memory and partitioned into subgraphs of size at most h, again using the algorithm of [1]. This introduces at most $\mathcal{O}\big(N/\sqrt{h}\big)$ additional separator vertices, so that S is a separator of size $\mathcal{O}\big(N/\sqrt{h}\big)$ that partitions G into subgraphs of size at most h.

Now let us analyze the I/O-complexity of this procedure. Computing the initial separator S_r of G_r takes $\mathcal{O}(|G_r|) = \mathcal{O}(N/B)$ I/Os, by Properties (i) and (v) of the graph hierarchy. To compute separator S_i from separator S_{i+1}, the algorithm has to identify the vertices in S_i', compute the connected components of $G_i - S_i'$, and load each of them into main memory, where it is partitioned into subgraphs of size at most $h\log^2 B$. The construction of the graph hierarchy can easily ensure that every vertex $v \in G_i$ is labelled with the vertex in G_{i+1} that represents v. Under this assumption vertex set S_i' can be identified in $\mathcal{O}(\text{sort}(|G_i|))$ I/Os by sorting and scanning the vertex set of G_i and the separator S_{i+1} a constant number of times. Computing the connected components of $G_i - S_i'$ takes $\mathcal{O}(\text{sort}(|G_i|))$ I/Os, by Theorem 5.11. Once the connected components of $G_i - S_i'$ have been computed, loading each of them into main memory to compute separator S_i'' takes $\mathcal{O}(\text{scan}(|G_i|))$ I/Os. The computation of separator S from separator S_0 is carried out in the same manner as the computation of separator S_i from separator S_i'. Hence, this takes $\mathcal{O}(\text{sort}(N))$ I/Os, and the total I/O-complexity of computing separator S from the graph hierarchy is $\mathcal{O}\big(N/B + \sum_{i=0}^{r-1}\text{sort}(|G_i|) + \text{sort}(N)\big) = \mathcal{O}\big(\sum_{i=0}^{r-1}\text{sort}(N/2^i) + \text{sort}(N)\big) = \mathcal{O}(\text{sort}(N))$. This proves the following lemma.

Lemma 9.1. *Given a graph hierarchy $G = G_0, G_1, \ldots, G_r$ with Properties (i)–(v) above, a separator S of size $\mathcal{O}\big(N/\sqrt{h}\big)$ that partitions G into subgraphs of size at most h can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, provided that $M \geq 56h\log^2 B$.*

9.2 Computing the Graph Hierarchy

What remains to be shown is how to compute the graph hierarchy. Since graph G is planar, and edge contractions preserve planarity, Property (ii) is guaranteed if graphs $G_1, \dots, G_r$ are constructed using edge contractions. The difficult part is ensuring Properties (iii)–(v) simultaneously. We first outline the basic approach taken and then argue how to perform this computation in an I/O-efficient manner.

Since graph $G_0 = G$ satisfies Properties (ii)–(v), we can assume that graphs $G_0, \dots, G_{i-1}$ are given and graph G_i has to be constructed from graph G_{i-1} through a series of edge contractions. In order to do that, let $\omega(v)$ and $\sigma(v)$ be two labels, for every vertex v in graphs $G_0, \dots, G_r$. Label $\omega(v)$ is the number of vertices in G represented by v and is called the *weight* of vertex v. Note that every vertex in G_0 has weight one. If $v \in G_i$, $i > 0$, then $\sigma(v)$ is the number of vertices in G_{i-1} represented by v; $\sigma(v)$ is called the *size* of v. In order to satisfy Properties (iii) and (iv), the algorithm ensures the following invariant:

(I) For every vertex in G_i, $\omega(v) \leq 2^i$ and $\sigma(v) \leq 56$.

The construction starts with a graph $G_i' = G_{i-1}$. Every vertex in G_i' has the same weight as in G_{i-1}. The size of every vertex in G_i' is one. A vertex $v \in G_i'$ is said to be *heavy* if either $\omega(v) > 2^{i-1}$ or $\sigma(v) > 28$. Otherwise v is *light*. An edge $(v, w) \in G_i'$ is *contractible* if both its endpoints are light. It is obvious that a contractible edge can be contracted while maintaining Invariant (I). The algorithm now contracts contractible edges until no such edge remains.

Let G_i'' be the graph obtained when no more contractions are possible. By the definition of a contractible edge, no two light vertices in G_i'' are adjacent. Now the light vertices of degree at most two are partitioned into maximal subsets so that the vertices in each subset are adjacent to the same set of heavy vertices, have total weight at most 2^i and total size at most 56. The light vertices in each such set are replaced by a single vertex. Let G_i be the graph obtained from G_i'' in this manner.

Lemma 9.2. *Graph G_i as constructed by the above procedure has Properties (ii)–(v).*

Proof. (ii): By induction, we can assume that $G_i' = G_{i-1}$ is planar. Hence, G_i'' is planar, as it is obtained from G_i' through a series of edge contractions. An embedding of G_i can be obtained from an embedding of G_i'' as follows: Let S be a set of light vertices in G_i'' represented by a single vertex v_S in G_i. Then choose a vertex $v \in S$ and remove all vertices in $S \setminus \{v\}$ from G_i''. Rename v to v_S.

(iii) and (iv): Graph G_i' has Property (iii) and, by induction, Property (iv). Hence, graph G_i can violate either of these two properties only if the construction merges a set of vertices whose total weight exceeds 2^i or whose total size exceeds 56. Since this is not done, graph G_i has Properties (iii) and (iv).

(v): To prove that graph G_i has Property (v), we make use of the following proposition, whose proof can be found in [32].

Proposition 9.3. *Let $G = (V_1, V_2, E)$ be a bipartite planar graph. Let vertex set V_2 be partitioned into non-empty equivalence classes $C_1, \dots, C_q$, where two vertices in V_2 are equivalent if they have degree at most two and are adjacent to the same set of vertices in V_1. Then $q \leq 6|V_1|$.*

Using this fact, we can prove that $|G_i| \leq 28N/2^i$. In particular, this claim is true for $G_0 = G$. So assume that the claim holds for all graphs $G_0, \dots, G_{i-1}$, and consider the subgraph H of G_i induced by the edges incident to light vertices. Graph H is bipartite, and no two light vertices of degree at most two in H are adjacent to the same set of heavy vertices. Hence, by Proposition 9.3, the number of light vertices in G_i is bounded by $6h_i$, and the total number of vertices in G_i is at most $7h_i$, where h_i is the number of heavy vertices in G_i. Thus, in order to prove that $|G_i| \leq 28N/2^i$, it suffices to show that $h_i \leq 4N/2^i$. To do this, we partition the heavy vertices in G_i into two classes: A vertex v of G_i is of type I if $\omega(v) > 2^{i-1}$. It is of type II if $\omega(v) \leq 2^{i-1}$, but $\sigma(v) > 28$. There are at most $N/2^{i-1}$ type-I vertices and at most $|G_{i-1}|/28$ type-II vertices. Hence,

$$\begin{aligned} h_i &\leq \frac{N}{2^{i-1}} + \frac{|G_{i-1}|}{28} \\ &\leq \frac{N}{2^{i-1}} + \frac{N}{2^{i-1}} \\ &= \frac{4N}{2^i}. \end{aligned}$$

□

By Lemma 9.2, the above strategy for constructing graph G_i from graph G_{i-1} guarantees that G_i has Properties (ii)–(v). Constructing G_i' from G_{i-1} is a matter of changing the size $\sigma(v)$ of every vertex to one. Hence, this takes $\mathcal{O}(\text{scan}(|G_{i-1}|))$ I/Os. To obtain graph G_i from graph G_i'', it suffices to sort the light vertices of degree at most two by their neighbors and then partition each equivalence class of light vertices into maximal groups of consecutive vertices of total weight at at most 2^i and total size at most 56. This takes $\mathcal{O}(\text{sort}(|G_i''|)) = \mathcal{O}(\text{sort}(|G_{i-1}|))$ I/Os. So let us concentrate on the construction of G_i'' from G_i'.

This construction has to be done with some care because the contraction of an edge $\{v, w\}$ may render another edge $\{v, w'\}$ non-contractible; but contracting the edges in G_i' one at a time, in order to check whether each edge to be contracted is contractible, does not seem to lead to an I/O-efficient algorithm. The solution to this problem is a strategy that iteratively contracts sets of edges that are guaranteed not to interfere with each other's contractibility. The contractions in each iteration are sufficient to guarantee that the minimum size of the vertices in the graph increases by a factor of two from one iteration to the next, so that only $\lceil \log_2 28 \rceil$ iterations are required before no contractible edges remain. The pseudo-code of this procedure is shown in Algorithm 9.1.

In this procedure the *contractible subgraph* of a graph G is the subgraph of G induced by the contractible edges in G. In each iteration, the algorithm can restrict its attention to graph H_j because the edges of G_i' that are not in H_j are not contractible. The contractions in each iteration are divided into two phases.

The first phase (Lines 4–5) contracts the edges in a maximal matching $\mathcal{M}$ of H_j. The contraction of any subset of the edges in $\mathcal{M}$ cannot affect the contractibility of the remaining edges in $\mathcal{M}$ because no two edges in $\mathcal{M}$ share an

Procedure COMPRESS

```
1: $H_0 \leftarrow$ contractible subgraph of $G'_i$
2: $j \leftarrow 0$
3: **while** $H_j \neq \emptyset$ **do**
4:   Compute a maximal matching $\mathcal{M}$ of $H_j$.
5:   Contract the edges in $\mathcal{M}$.
6:   **for** every unmatched vertex $v$ in $H_j$ **do**
7:     **if** $v$ has a light matched neighbor $w$ **then**
8:       Contract $v$ into $w$.
9:     **end if**
10:  **end for**
11:  $H_{j+1} \leftarrow$ contractible subgraph of $H_j$
12:  $j \leftarrow j + 1$
13: **end while**
```

Algorithm 9.1
Computing graph G''_i from graph G'_i.

endpoint. Hence, this simultaneous contraction of the edges in $\mathcal{M}$ does not contract an edge that would have become non-contractible when performing edge contractions one at a time.

After this first phase, the vertices of H_j can be partitioned into two categories: A *matched* vertex represents the two endpoints of an edge in $\mathcal{M}$. All other vertices are *unmatched.* The goal of the second phase (Lines 6–10) is to ensure that the vertex set of graph H_{j+1} contains only matched vertices of H_j, i.e., that no remaining unmatched vertex in H_j has an incident edge that is contractible. It is easy to show that this implies that every vertex in H_j has size at least 2^j, so that the procedure terminates after at most $\lceil \log_2 28 \rceil$ iterations of the while-loop, and the size of graph H_j is at most $|G'_i|/2^j$.

To eliminate all unmatched vertices from H_j in an I/O-efficient manner, observe that the maximality of matching $\mathcal{M}$ implies that all neighbors of an unmatched vertex are matched. Hence, the algorithm has to solve a bipartite contraction problem where the set of matched vertices is fixed an every unmatched vertex should be contracted into one of its matched neighbors if possible. this can be done as follows: Denote the set of matched and unmatched vertices of H_j by V_m and V_u, respectively, and assume that every vertex in H_j has a unique numerical ID. Then construct a DAG D with vertex set V_m. For every vertex $v \in V_u$, graph D contains a path $P_v = (w_1, \dots, w_k)$, where $w_1, \dots, w_k$ are the neighbors of v in H_j, sorted by increasing numbers. Now use time-forward processing to pass every vertex $v \in V_u$ along its path P_v in D. Every vertex $w \in V_m$ inspects the unmatched vertices $v_1, \dots, v_l$ it receives from its in-neighbors. Let $0 \leq h \leq l$ be the minimum index so that $\omega(w) + \sum_{a=1}^{h} \omega(v_a) > 2^{i-1}$ and $\sigma(w) + \sum_{a=1}^{h} \sigma(v_a) > 28$. If no such index exists, let $h = l$. Then vertices $v_1, \dots, v_h$ are marked for contraction into vertex w. After these contractions vertex w is heavy, so that edges $\{v_{h+1}, w\}, \dots, \{v_l, w\}$ are not contractible. Hence, vertices $v_{h+1}, \dots, v_l$ are forwarded to the out-neighbors of w on paths

$P_{v_{h+1}}, \dots, P_{v_l}$, to test whether they can be contracted into those vertices. Once graph D has been processed, the vertices in V_u that have been marked for contraction into a vertex in V_m can be contracted into these matched vertices using the standard graph contraction procedure.

This procedure achieves the desired result because every vertex $v \in V_u$ that is not contracted into one of its neighbors is passed along the whole path P_v in D, and every edge $\{v, w\}$, $w \in P_v$ is tested for its contractibility. Hence, if one of these edges were contractible, it would have been contracted.

One iteration of the procedure COMPRESS takes $\mathcal{O}(\text{sort}(|H_j|))$ I/Os: A maximal matching of H_j can be computed in this number of I/Os [25, 32] (see the remark at the end of Section 4.2). The contraction of the edges in $\mathcal{M}$ can be carried out in $\mathcal{O}(\text{sort}(|H_j|))$ I/Os in the standard fashion. The construction of DAG D from H_j requires sorting and scanning the vertex and edge sets of H_j a constant number of times. DAG D has size $\mathcal{O}(|H_j|)$, so that the application of time-forward processing to D takes $\mathcal{O}(\text{sort}(|H_j|))$ I/Os (see Section 4.1). Contracting the marked unmatched vertices into their matched neighbors takes another $\mathcal{O}(\text{sort}(|H_j|))$ I/Os using the standard contraction procedure.

We have shown that one iteration of Algorithm 9.1 takes $O(\text{sort}(|H_j|))$ I/Os. We have also argued that $|H_j| \leq |G_i'|/2^j$. This implies that the total I/O-complexity of procedure COMPRESS is $\mathcal{O}(\text{sort}(|G_i'|)) = \mathcal{O}(\text{sort}(|G_{i-1}|))$. Since the construction of graph G_i from graph G_i'' also takes $\mathcal{O}(\text{sort}(|G_{i-1}|))$ I/Os, the whole construction of graph G_i from graph G_{i-1} takes $\mathcal{O}(\text{sort}(|G_{i-1}|))$ I/Os. By Property (v) of the graph hierarchy, this shows the following lemma.

Lemma 9.4. *A graph hierarchy $G = G_0, G_1, \dots, G_r$ with Properties (i)–(v) can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

The following theorem now follows immediately from Lemmas 9.1 and 9.4.

Theorem 9.5. *Given a planar graph $G = (V, E)$ and an integer $h > 0$, a separator S partitioning G into subgraphs of size at most h can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, provided that $M \geq 56h \log^2 B$. The size of S is $\mathcal{O}(N/\sqrt{h})$.*

10 Planar Undirected DFS

As the last result discussed in detail in this course, we now return to DFS in undirected graphs. This time, however, we restrict our attention to undirected planar graphs. As before we assume that the graph is given together with an embedding, and we do not distinguish between a graph and its embedding.

The algorithm we discuss is due to Arge *et al.* [5]. Similar ideas have been used in a PRAM-algorithm for DFS in planar graphs by Hagerup [16]. The algorithm combines two ideas: The first one is that DFS in any graph can be reduced to computing appropriate DFS-trees of its biconnected components and "gluing" them together. The main idea for constructing a DFS-tree of a biconnected planar graph G is to partition G into layers of extremely simple structure, using BFS in a graph that is closely related to the dual of G. In particular, these layers

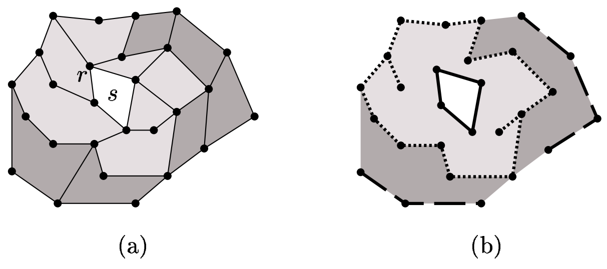

Figure 10.1
(a) A partition of the faces of G into levels. (b) The layers defined by this partition.

are trees of cycles. A DFS-tree of a tree of cycles can be obtained by computing the biconnected components of the graph (i.e., the cycles) and removing an appropriate edge from each component. Moreover, the relationship between these layers and the structure of G is such that a DFS-tree of G can be obtained by "gluing" together appropriate DFS-trees of the layers.

10.1 Partitioning the Graph into Layers

Formally, the layers of G are defined as follows (see Figure 10.1): Let r be the source of the DFS, i.e., the root of the DFS-tree to be computed. Let s be a face of G that has r on its boundary. Then the faces of G are partitioned into levels as follows: Face s is the only level-0 face. A face is at level $i > 0$ if it shares a vertex with a face at level $i-1$, but not with a face at level less than $i-1$. Given the levels of the faces of G, the level of a vertex or edge x is defined as the minimum level of the faces that have x on their boundaries. Let V_i be the set of vertices at level i. Then V_i is the vertex set of layer L_i. An edge e is an edge of layer L_i if it is at level i and both its endpoints are at level i. Denote the set of these edges by E_i. That is, $L_i = (V_i, E_i)$. Finally, an edge at level i that has at least one endpoint at level $i-1$ is called an *attachment edge* of layer L_i. In particular, such an edge connects a vertex in L_i with a vertex in L_{i-1} or two vertices in L_{i-1}. Let A_i be the set of attachment edges of layer L_i.

Before showing that layers $L_0, \dots, L_k$ have a very simple structure, we argue that these layers and their sets of attachment edges can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os using procedure LayerPartition outlined in Algorithm 10.1. We do not discuss every single detail of the algorithm, but present the main ideas.

The face-on-vertex graph G_F computed in Line 1 of the algorithm is defined as follows (see Figure 10.2): Graph G_F contains all vertices of G as well as one vertex f^* for every face f of G. There is an edge (v, f^*) in G_F if and only if vertex v is on the boundary of face f. We leave it as an exercise to verify that with this definition of G_F, the levels of the vertices and edges in G are computed correctly in Lines 4 and 5, and that sorting the vertex and edge sets of G as in

Procedure LayerPartition

```
1: Compute the face-on-vertex graph $G_F$ of $G$.
2: Choose a vertex $s^*$ in $G_F$ adjacent to vertex $r$.
3: Perform BFS in $G_F$ from $s^*$.
4: Let the level of every vertex $v \in G$ be $(d(s^*, v) - 1)/2$.
5: Let the level of every edge $e \in G$ be the minimum of $d(s^*, f_1^*)/2$ and $d(s^*, f_2^*)/2$,
   where $f_1$ and $f_2$ are the two faces that have $e$ on their boundaries.
6: Sort the vertices in $V$ by their levels to partition them into vertex sets $V_0, \dots, V_k$.
7: Sort the edges in $E$ by their levels as primary key and by the minimum of the
   levels of their endpoints as secondary key. This produces a partition of $E$ into sets
   $E_0, A_1, E_1, \dots, A_k, E_k$.
```

Algorithm 10.1
An algorithm to partition G into layers.

Lines 6 and 7 does indeed produce the desired partition of these sets into the vertex and edge sets of layers $L_0, \dots, L_k$ and the sets $A_1, \dots, A_k$ of attachment edges.

Assuming that every edge e of G "knows" the two faces f_1 and f_2 that have e on their boundaries, the computation of the levels of all vertices and edges of G requires a constant number of sort and scan operations and hence takes $\mathcal{O}(\text{sort}(N))$ I/Os. Lines 6 and 7 sort sets V and E and hence also take $\mathcal{O}(\text{sort}(N))$ I/Os. Thus, the main difficulty of the algorithm is the computation of graph G_F and performing BFS in G_F. The construction of graph G_F also provides every vertex and edge in G with the names of its two adjoining faces, thereby providing the computation in Line 5 with the required input.

In order to perform BFS in G_F, observe that G_F is obviously planar and has $\mathcal{O}(N)$ vertices. Hence, the shortest path algorithm from Section 8 can be used to compute a BFS-tree of G_F in $\mathcal{O}(\text{sort}(N))$ I/Os. What remains to be shown is how graph G_F can be constructed: First compute a set of cycles C_f, one per face f of G, so that cycle C_f contains one vertex per edge on the boundary of face f and the vertices appear in the same order along C_f as their corresponding edges clockwise around f. The collection of these cycles can be obtained from G using an adaptation of the Euler tour technique (see Section 3.1). In particular, replace every edge $\{v, w\} \in G$ with two directed edges (v, w) and (w, v) and define the successor of every edge (u, v) as edge (v, w) so that edges $\{v, u\}$ and $\{v, w\}$ appear consecutively in counterclockwise order around v. The graph G' defined as the union of cycles C_f is obviously planar. Hence it is sparse, and its connected components can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, by Theorem 5.11. The connected components of G' are the cycles C_f, and every vertex in C_f represents an edge (v, w). Now sort and scan the vertex set of G' and add a vertex f^* per cycle C_f and an edge $\{f^*, v\}$ per vertex (v, w) in cycle C_f to G_F. This takes another $\mathcal{O}(\text{sort}(N))$ I/Os.

Since all steps of Algorithm 10.1 can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os, we obtain the following lemma.

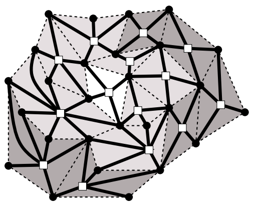

Figure 10.2
The face-on-vertex graph of graph G shown in Figure 10.1a.

Lemma 10.1. *A partition of an undirected planar graph G with N vertices into layers $L_0, \ldots, L_k$ and sets $A_1, \ldots, A_k$ of attachment edges can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, provided that $M = \Omega(B^2 \log^2 B)$.*

Remark. The above construction does not construct a planar embedding of G_F from the planar embedding of G; but the shortest path algorithm of Section 8 requires a planar embedding of G_F in order to perform BFS in G_F. Given that cycles C_f have been identified, graph G' can be transformed into a collection of linked lists, by removing one edge from each cycle. Now list ranking can be applied to determine the order of the edges clockwise around each face. This information suffices to construct a planar embedding of G_F.

10.2 DFS in a Layer

Given a partition of G into layers as computed by procedure LayerPartition, we now focus on a single layer L_i and show that it has a sufficiently simple structure to perform DFS in L_i I/O-efficiently. In particular, we say that a biconnected component of L_i is *trivial* if it consists of a single edge. Otherwise the biconnected component is *non-trivial*. Let G_i be the subgraph of G induced by the faces at levels 0 through i. We call a cycle in G_i a *boundary cycle* if the incident faces on one side of the cycle are at level i, while the incident faces on the other side are at level $i + 1$.

Lemma 10.2. *The non-trivial biconnected components of L_i are the boundary cycles of G_i.*

Proof. Consider a cycle C in L_i. All faces incident to C are at level i or $i + 1$. The faces of G at level at most $i - 1$ form a connected region. Hence, either all these faces are inside C, or all of them are outside C. This implies that either all faces outside C or all faces inside C are at level at least $i + 1$ because they cannot share a vertex with a level-$(i-1)$ face. This proves that every cycle in L_i is a boundary cycle.

Every non-trivial biconnected component of L_i that is not a cycle contains two vertices v and w so that there are three internally vertex-disjoint paths P_1,

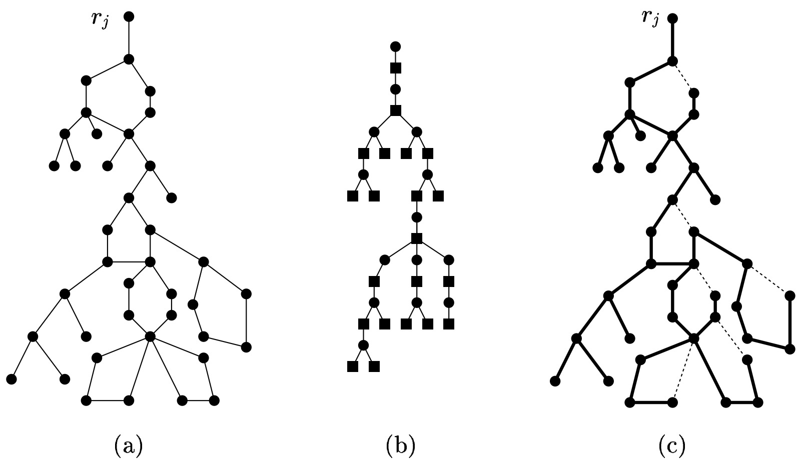

Figure 10.3
(a) A tree G of cycles. (b) The corresponding block-cutpoint-tree. Block nodes are squares; cutpoints are discs. (c) A DFS-tree of G. Dotted edges are non-tree edges.

P_2, and P_3 from v to w in L_i. These paths define two cycles $P_1 \cup P_2$ and $P_1 \cup P_3$, which are both boundary cycles. However, this is impossible because either P_3 is completely inside or completely outside the region bounded by $P_1 \cup P_2$. □

Now let $H_1, \ldots, H_q$ be the connected components of L_i, and let $r_1, \ldots, r_q$ be vertices so that $r_i \in H_i$. We describe a procedure that uses Lemma 10.2 to compute DFS-trees $T_1, \ldots, T_q$ of $H_1, \ldots, H_q$ rooted at vertices $r_1, \ldots, r_q$.

In order to compute one such DFS-tree T_j, compute the block-cutpoint-tree T'_j of H_j (see Figure 10.3). Tree T'_j contains all cutpoints of H_j and one vertex per biconnected component of H_j. If vertex r_j is not a cutpoint of H_j, it is added as a vertex to T'_j. There is an edge $\{v, \beta\}$ in T'_j, where β represents a biconnected component $\mathcal{B}$ of H_i, if vertex v is contained in $\mathcal{B}$. Choose vertex r_i as the root of T'_j. The parent cutpoint of a biconnected component is defined as the parent of the corresponding node in T'_j. A DFS-tree T_j of H_j can now be obtained by removing one of the two edges incident to its parent cutpoint from every non-trivial biconnected component of H_j. Next we show that tree T_j is indeed a DFS-tree of H_j and that the construction of trees $T_1, \ldots, T_q$ can be carried out I/O-efficiently.

Lemma 10.3. *Given a layer L_i with connected components $H_1, \ldots, H_q$ and a set of vertices $r_1, \ldots, r_q$ so that $r_j \in H_j$, a set of DFS-trees $T_1, \ldots, T_q$ for graphs $H_1, \ldots, H_q$ rooted at vertices $r_1, \ldots, r_q$ can be computed in $\mathcal{O}(\text{sort}(|L_i|))$ I/Os.*

Proof. First we show that tree T_j as constructed by the above procedure is a DFS-tree of H_j. To do this, we consider a biconnected component $\mathcal{B}$ of H_j

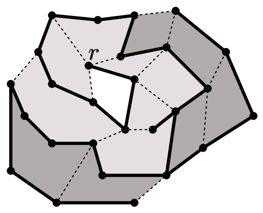

Figure 10.4
A DFS-tree of G.

containing a non-tree edge $\{v, w\}$. One of the endpoints of this edge, say v, is the parent cutpoint of $\mathcal{B}$. Hence, any path from r_j to w in H_j must contain v. In particular, this is true for the path from r_j to w in T_j, so that v is an ancestor of w in T_j. Since this is true for any non-tree edge $\{v, w\}$, T_j is a DFS-tree of H_j.

Next we prove that the computation of tree T_j for graph H_j can be carried out in $\mathcal{O}(\text{sort}(|H_j|))$ I/Os, which implies that the computation of trees $T_1, \dots, T_q$ takes $\mathcal{O}\big(\sum_{j=1}^{q} \text{sort}(|H_j|)\big) = \mathcal{O}(\text{sort}(|L_i|))$ I/Os. The biconnected components of H_j can be computed in $\mathcal{O}(\text{sort}(|H_j|))$ I/Os, using the algorithm from Section 5.3. We leave it as an exercise to verify that given the biconnected components of H_j, the block-cutpoint-tree T'_j can be computed in $\mathcal{O}(\text{sort}(|H_j|))$ I/Os, by sorting and scanning the vertex and edge sets of H_j a constant number of times. Then the Euler tour technique and list ranking can be applied to root T'_j at r_j and determine the parent cutpoint of every biconnected component. Given the parent cutpoint of every biconnected component, it suffices to scan the edge set of that biconnected component to (a) decide whether it is non-trivial (i.e., has more than one edge) and if so, (b) find one of the two edges incident to the parent cutpoint and remove it. $\square$

The construction outlined above computes a DFS-tree T_j for graph H_j. In order to use this tree in the construction of the next section, every vertex has to be labelled with its distance from r_j in T_j. This can be done in $\mathcal{O}(\text{sort}(|H_j|))$ I/Os, using the Euler tour technique and list ranking again (see Section 3).

10.3 DFS in a Biconnected Planar Graph

Having developed a tool for constructing DFS-trees of the layers of G, we now show how to obtain a DFS-tree of a biconnected planar graph from appropriate DFS-trees of its layers. In particular, the DFS-algorithm starts with a DFS-tree T_0 of $G_0 = L_0$ and then iteratively augments the current DFS-tree T_i of G_i with DFS-trees of the connected components of L_{i+1} to obtain a DFS-tree T_{i+1} of G_{i+1}. A DFS-tree for G obtained in this manner is shown in Figure 10.4. If we can show that the augmentation can be carried out in $\mathcal{O}(\text{sort}(|L_i|+|L_{i+1}|))$, it follows that the whole algorithm takes $\mathcal{O}(\text{sort}(N))$ I/Os because layers $L_0, \dots, L_k$ are disjoint.

Tree T_0 is easy to obtain using the Euler tour technique and list ranking because graph G_0 is a simple cycle.

So assume that a DFS-tree T_i of G_i is given, which is to be augmented to produce a DFS-tree T_{i+1} of G_{i+1}. Let $\bar{G}_i$ be the subgraph of G induced by all faces at level at least $i + 1$. Since the faces at levels 0 through i form a connected region, the boundary between G_i and $\bar{G}_i$ is a collection of edge-disjoint simple cycles and the removal of the faces of $\bar{G}_i$ introduces a number of "holes" $R_1, \dots, R_t$ whose boundaries are the boundary cycles of G_i. By Lemma 10.2, these boundary cycles are the biconnected components of L_i. The following observation now follows immediately from the way the DFS-trees for the connected components of L_i are constructed.

Observation 10.1. *Let R_j be a hole of G_i, and let $v_1, \dots, v_k$ be the vertices on its boundary, sorted clockwise around R_j and so that v_1 has minimum depth in T_i. Then v_1 is an ancestor of vertices $v_2, \dots, v_k$, and either $(v_1, \dots, v_k)$ or $(v_1, v_k, \dots, v_2)$ is a path in T_i.*

Intuitively, if w.l.o.g. $(v_1, \dots, v_k)$ is the path in T_i, the observation states that for any vertex v_i, vertices $v_1, \dots, v_{i-1}$ are ancestors of v_i in T_i. Hence, the following strategy produces a DFS-tree for G_{i+1}: For every connected component H_j of L_i, find the set A'_j of attachment edges of H_j. Every edge in A'_j has one endpoint on the boundary of the hole R containing H_j and the other endpoint in H_j. Find the attachment edge $\{u_j, v_j\}$ whose endpoint u_j on the boundary of R has maximal depth. Then compute a DFS-tree of H_j rooted at v_j and link it to T_i using edge $\{u_j, v_j\}$. Let T_{i+1} be the tree obtained by attaching DFS-trees for all connected components of L_{i+1} to T_i in this manner.

Lemma 10.4. *Tree T_{i+1} is a DFS-tree of G_{i+1}.*

Proof. We have to show that for every non-tree edge $\{v, w\}$ of T_{i+1} w.l.o.g. v is an ancestor of w. We distinguish three cases: (1) $v, w \in G_i$, (2) $v \in G_i$ and $w \in L_{i+1}$, and (3) $v, w \in L_{i+1}$. For Cases (1) and (3) the claim holds because T_{i+1} is the union of a DFS-tree T_i for G_i and DFS-trees for the connected components of L_{i+1}.

In Case (2) let $w \in H_j$. Then v is on the boundary of the hole containing H_j. In particular, by the choice of the attachment edge $\{u_j, v_j\}$ of H_j included in T_{i+1}, v is an ancestor of u_j in T_i. Vertex w is a descendant of v_j in the DFS-tree constructed for H_j. This implies that v is an ancestor of w in T_{i+1}. $\square$

Now observe that the above construction requires little more than a constant number of sort and scan operations. In particular, the connected components of L_{i+1} can be found in $\mathcal{O}(\text{sort}(|L_{i+1}|))$ I/Os, by Theorem 5.11. Given the connected components $H_1, \dots, H_j$, it suffices to sort the set A_{i+1} of attachment edges of L_{i+1} by their endpoints in L_{i+1}, sort the vertices in L_{i+1} by their numbers, and scan the two sorted lists to determine for every attachment edge the connected component H_j of L_{i+1} containing one of its endpoints. After sorting the vertices in L_i by their IDs and the attachment edges in A_{i+1} by their endpoints in L_i, a single scan of these two sorted lists suffices to label every attachment edge of L_{i+1} with the depth of its endpoint in T_i. Now sort the attachment edges of L_{i+1} by the connected components of L_{i+1} containing one of their endpoints as the primary key and by the depths of their endpoints in T_i as the secondary key. This produces sets A'_j, each with its edges sorted by increasing depths of their endpoints in T_i. A single scan of these sorted lists suffices to extract edge $\{u_j, v_j\}$ as the first edge in A'_j, for every connected component H_j. In order to construct the DFS-trees for $H_1, \dots, H_q$, the construction of the previous section is used. Clearly this procedure takes $\mathcal{O}(\mathrm{sort}(|L_i| + |L_{i+1}|))$ I/Os. Hence, we obtain the following result.

Lemma 10.5. *A DFS-tree of a biconnected planar graph G with N vertices can be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os, provided that $M = \Omega(B^2 \log^2 B)$.*

10.4 DFS in Connected Planar Graphs

Finally, we are ready to put the bits and pieces together to obtain a DFS-algorithm for connected planar graphs. In fact, the algorithm just uses ideas already presented above: If the graph is biconnected, apply Lemma 10.5 to obtain a DFS-tree of G. If G is not biconnected, apply a similar procedure as for DFS in a layer. In particular, compute the biconnected components of G, build the corresponding block-cutpoint-tree, and construct for every biconnected component of G, a DFS-tree rooted at its parent cutpoint. Since every non-tree edge has both its endpoints in the same biconnected component, it is obvious that the union of these DFS-trees is a DFS-tree of G.

The computation of the biconnected components takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os using the biconnectivity algorithm from Section 5.3. Computing a DFS-tree for a biconnected component of size N_i takes $\mathcal{O}(\mathrm{sort}(N_i))$ I/Os, by Lemma 10.5. Since the total size of all biconnected components is $\mathcal{O}(N)$, computing DFS-trees for all biconnected components therefore takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, and we obtain the following result.

Theorem 10.6. *A DFS-tree of a connected planar graph with N vertices can be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os, provided that $M = \Omega(B^2 \log^2 B)$.*

11 Lower Bounds

So far we have focused on the design of I/O-efficient algorithms for fundamental graph problems. In this section we try to answer the question whether these algorithms are optimal or close to optimal by proving lower bounds for some of the problems solved by the algorithms presented in Sections 2–10.

In order to prove these lower bounds, we concentrate on two central problems: list ranking and connected components. Once we have shown that these problems require $\Omega(\mathrm{perm}(|V|))$ and $\Omega(\mathrm{perm}(|E|))$ I/Os, the same lower bounds can be obtained for numerous other problems using rather simple arguments.

Before going into the details of the proofs, a few remarks regarding the choice of an appropriate model of computation are in order because choosing the right model for proving lower bounds for graphs problems is non-trivial. Consider for example two common models assumed in lower bound proofs. The first model assumes that records are indivisible. That is, the output has to be represented as an appropriate permutation of the input because the model does not allow the creation of new records. This model is too restrictive because most interesting graph problems require the computation of a labelling of the vertices of the graph, so that any algorithm for this problem is forced to create new records representing the computed labels. The second model is the comparison model, which in particular does not allow any indirect addressing (i.e., exploiting the fact that computers represent everything as numbers, which allows the use of data items as indices for accesses into arrays). But internal memory graph algorithms make extensive use of indirect addressing, so that disallowing it in I/O-efficient algorithms may overly handicap the latter and therefore prevent a meaningful comparison between internal and external memory algorithms for the same problem. Choosing much more powerful models, on the other hand, makes it hard to prove non-trivial lower bounds.

The lower bound proof for list ranking presented in Section 11.1 assumes that records are indivisible, which requires some care when formulating the arguments in the proofs. In particular, the constructions presented in the proofs could be considered reductions from one problem to another. But these reductions would create new records and thereby leave the model. Instead we emphasize that the constructions are not carried out by an algorithm, but we use them only as tools to prove the equivalence between input instances for the two problems. Assuming indivisibility of records also implies that the arguments apply only to a particular type of algorithm, which we specify carefully.

The lower bound proof for connectivity uses an augmented version of the comparison model: the indexed I/O-tree [28]. Essentially this model is the comparison model augmented with indirect addressing. The details of the model are of less relevance to our argument here because we use a reduction that requires a very weak model; but the lower bound of the problem we reduce to connectivity to prove a lower bound for connectivity is shown in the indexed I/O-tree model.

11.1 List Ranking, BFS, DFS, and Shortest Paths

A lower bound for list ranking can be obtained by showing its equivalence to the *split proximate neighbors* (SPN) problem. In this problem, a sequence S of $2N$ integers in the range 1 through N is given. Sequence S is the concatenation of two sequences S_1 and S_2 of length N so that each integer occurs exactly once in S_1 and exactly once in S_2. Sequence S_1 is sorted. The goal is to permute the elements in S so that for every integer $1 \leq i \leq N$, both occurrences of i in S are stored in the same disk block.

The original lower bound proof for this problem [3] proves the lower bound by counting the number of different permutations an algorithm solving SPN has to be able to produce. Here we present a more intuitive proof.

Lemma 11.1. *The split proximate neighbor problem requires $\Omega(\mathrm{perm}(N))$ I/Os for an input sequence of size $2N$.*

Proof. To prove the lemma, we show that if there is an algorithm that solves SPN in $\mathcal{I}(N)$ I/Os, there is an algorithm that can permute N data items in $\mathcal{O}(\mathcal{I}(N))$ I/Os. Hence, $\mathcal{I}(N) = \Omega(\mathrm{perm}(N))$.

So let $x_1, \ldots, x_N$ be a set of data items, and let $\sigma : [1, N] \to [1, N]$ be a permutation so that elements $x_1, \ldots, x_N$ have to be arranged in the order $x_{\sigma(1)}, \ldots, x_{\sigma(N)}$. Let $y = (1, 2, \ldots, N, \sigma(1), \sigma(2), \ldots, \sigma(N))$ be the instance of SPN defined by permutation σ. Now consider an algorithm $\mathcal{A}$ that solves SPN in $\mathcal{I}(N)$ I/Os, let $\mathcal{I}(y) \leq \mathcal{I}(N)$ be the number of I/Os performed by algorithm $\mathcal{A}$ on instance y, and let S be the sequence of data moves performed by algorithm $\mathcal{A}$ on instance y. That is, sequence S incurs $\mathcal{I}(y)$ I/Os. The elements $x_1, \ldots, x_N$ can be arranged in the order $x_{\sigma(1)}, \ldots, x_{\sigma(N)}$ in at most $2\mathcal{I}(y) \leq 2\mathcal{I}(N)$ I/Os as follows: First apply the same movements to elements $x_1, \ldots, x_N$ as algorithm $\mathcal{A}$ applies to elements $1, 2, \ldots, N$. Now reverse the data movements of algorithm $\mathcal{A}$, letting element $x_{\sigma(i)}$ play the role of element $\sigma(i)$ in y. To do this, element $x_{\sigma(i)}$, $1 \leq i \leq N$, has to be moved into the place of element $\sigma(i)$ before running algorithm $\mathcal{A}$ backwards. However, after running algorithm $\mathcal{A}$ forward, element $x_{\sigma(i)}$ is stored in the same block into which algorithm $\mathcal{A}$ places element $\sigma(i)$. Hence, element $x_{\sigma(i)}$ can be moved into the place of element $\sigma(i)$ when the reversal of algorithm $\mathcal{A}$ loads the block containing element $x_{\sigma(i)}$ into main memory for the first time. This does not incur any extra I/Os.

Now let $\mathcal{A}'$ be an algorithm that behaves as just described for any input instance $x_1, \ldots, x_N$ and any permutation σ. Since the above construction does not make any assumptions about the structure of permutation σ, algorithm $\mathcal{A}'$ arranges any input instance in the correct order and does so in at most $2\mathcal{I}(N)$ I/Os. Given the remark at the beginning of the proof, this proves the lemma. □

Lemma 11.1 can be used to prove a lower bound on the number of I/Os performed by algorithms that are able to solve list ranking in its full generality as stated in Section 2. In particular, we restrict our attention to algorithms that solve the weighted list ranking problem using only the associativity of the sum operator defined on the set of vertex labels in the list. Note that this means that the lower bound does not hold for the unweighted list ranking problem because $(\mathbb{Z}, +)$ is a group, so that some clever algorithm for this problem may combine addition and subtraction to compute the ranks of all nodes more efficiently.

Given that the algorithm uses only the associativity of summation, it can be enforced that for every node x_i, there is some point during the course of the algorithm when nodes x_i and $\mathrm{succ}(x_i)$ reside in main memory together. If the algorithm does not already have this property, it can be enforced at the expense of increasing the I/O-complexity of the algorithm by only a constant factor.

Now every SPN instance gives rise to an equivalent list ranking instance. In particular, the successor of element i in sequence S_1 is defined to be element i in sequence S_2. The successor of element i in S_2 is element $i + 1$ in S_1. Consider the I/Os performed by the list ranking algorithm. Whenever two equal elements from S_1 and S_2 end up in main memory at the same time, they can be moved to a buffer of size B, which is emptied to disk whenever it runs full. The resulting algorithm performs N/B I/Os more than the list ranking algorithm and solves SPN. Since SPN requires $\Omega(\text{perm}(N))$ I/Os, we obtain the following result.

Theorem 11.2. *List ranking requires* $\Omega(\text{perm}(N))$ *I/Os.*

As an immediate consequence of Theorem 11.2, we obtain lower bounds for BFS, DFS, and SSSP. In particular, it suffices to consider the given list as an undirected graph whose edges have unit weights. Then list ranking can be solved by performing BFS, DFS or SSSP in this graph, starting at the head of the list. Again the lower bound applies only to algorithms that compute the distances of the vertices of G from the source only by adding path lengths.

Corollary 11.3. *Breadth-first search, depth-first search and single source shortest paths require* $\Omega(\text{perm}(N))$ *I/Os on a graph with* N *vertices.*

11.2 Connected and Biconnected Components

In order to prove a lower bound for the number of I/Os required to compute the connected components of a graph, we use the following proposition shown in [28]. Let the *segmented duplicate elimination problem* be defined as follows: Let S be a set of N integers drawn from the interval $[P+1, 2P]$, and let $P < N < P^2$. Furthermore, assume that S can be divided into P contiguous subsequences $S_1, \ldots, S_P$, each of length N/P, so that the elements in each sequence S_i are distinct. Then construct a Boolean array $C[P+1, \ldots, 2P]$ so that $C[i] = 1$ if and only if S contains an element of value i.

Proposition 11.4. *The segmented duplicate elimination problem with parameters* P *and* N *as above requires* $\Omega((N/P)\text{perm}(P))$ *I/Os.*

In order to prove an $\Omega(\text{perm}(|E|))$ lower bound for computing the connected components of a graph, the segmented duplicate elimination problem is reduced to that of computing the connected components of an appropriate graph. In particular, consider an instance of the segmented duplicate elimination problem with N elements in the range $[P+1, 2P]$, where $N \geq 2P$. Then graph G is defined as follows:

1. Graph G has $N/P + P$ vertices.
2. If $P + i \in S_j$, then G contains edge $\{j, N/P + i\}$.
3. Graph G contains edges $\{1,2\}, \{2,3\}, \ldots, \{N/P - 1, N/P\}$.

Graph G has $N/P + P = \Theta(P)$ vertices and $N + N/P - 1 = \Theta(N)$ edges. The construction of the edge set of G can easily be carried out in $\mathcal{O}(\text{scan}(N))$ I/Os. Now it is easy to see that $P+i \in S$ if and only if vertices 1 and $N/P+i$ are in the same connected component. Hence, computing the connected components of G requires $\Omega((N/P)\text{perm}(P)) = \Omega((|E|/|V|)\text{perm}(|V|)) = \Omega(\text{perm}(|E|))$ I/Os, and we obtain the following result.

Theorem 11.5. *Computing the connected components of a graph $G = (V, E)$ requires $\Omega(\text{perm}(|E|))$ I/Os.*

Using a similar construction, the same lower bound can be shown for computing the biconnected components of a graph. In particular, graph G above is augmented with a vertex 0 that is connected to vertex 1 and to vertices $N/P + 1, \dots, N/P + P$. Then element $P + i \in S$ if and only if vertices 0 and $N/P + i$ are in the same biconnected component of the augmented graph G. The augmentation can be carried out in $\mathcal{O}(\text{scan}(N))$ I/Os. The sizes of the vertex and edge sets of G remain $\Theta(P)$ and $\Theta(N)$, respectively. Hence, we obtain the following result.

Theorem 11.6. *Computing the biconnected components of a graph $G = (V, E)$ requires $\Omega(\text{perm}(|E|))$ I/Os.*

12 More Problems and Solutions

This last section is dedicated to a short survey of a few results that should not be missing from a course on I/O-efficient graph algorithms and a discussion of open problems related to the material presented in this course. In Section 12.1 we discuss three classes of sparse graphs other than planar graphs for which $\mathcal{O}(\text{sort}(N))$ I/O algorithms for BFS, DFS and the single source shortest path problem exist. In Section 12.2 we discuss the main ideas behind an I/O-efficient algorithm for planarity testing and planar embedding proposed in [26]. The algorithm is particularly interesting because it uses separators to compute the embedding, which is possible only because the separator algorithm from Section 9 does not use any information provided by a planar embedding of the graph. In Section 12.3 we discuss a number of interesting open problems.

12.1 More Classes of Sparse Graphs

There are a few more classes of sparse graphs that researchers have considered, trying to develop I/O-efficient algorithms for fundamental problems on these classes of graphs. The interest in these classes of sparse graphs stems either from their practical importance or from structural properties that made these graphs promising candidates for I/O-efficient solutions to the problems of interest. We start our discussion with the most practical class whose favorable structural properties are obvious to the trained eye. Then we work our way to graph classes whose practical relevance is disputable, but whose structure is more interesting.

Grid graphs. In [6] Arge *et al.* study problems on grid graphs. The vertices of a grid graph are a subset of the vertices of a regular $\sqrt{N} \times \sqrt{N}$ grid. Every vertex v can be connected to at most eight other vertices, namely the vertices whose grid positions differ by at most one in each dimension from the position of v. These graphs arise naturally in computations on raster-based elevation models used in geographic information systems.

An interesting fact to observe about grid graphs is that they are almost planar. That is, only diagonals can intersect and every diagonal intersects at most one other diagonal. Thus, it is not surprising that these graphs have small separators and that these separators can be used to compute shortest paths in the same way as for planar graphs.

In particular, choosing every B-th row and column to be in the separator, one obtains a separator of size $\mathcal{O}(N/B)$ that partitions the graph into $\mathcal{O}(N/B^2)$ subgraphs of size at most B^2 and boundary size $\mathcal{O}(B)$. Moreover, every separator vertex is on the boundary of at most four regions and the number of boundary sets is $\mathcal{O}(N/B^2)$. Hence, the shortest path algorithm for planar graphs can be applied to grid graphs, using the separator just defined instead of the separator computed for planar graphs using the algorithm from Section 9.

Depth-first search on grid graphs can be solved in $\mathcal{O}(N/\sqrt{B})$ I/Os using a slightly modified version of the internal memory DFS-algorithm. In particular, choosing the space between separator rows and columns to be $\sqrt{B}$, one obtains a separator of size $\mathcal{O}(N/\sqrt{B})$ that partitions the graph into $\mathcal{O}(N/B)$ subgraphs of size at most B and boundary size $\mathcal{O}(\sqrt{B})$. Now whenever the DFS-algorithm explores an edge connecting a separator vertex with an internal vertex of a subgraph G_i, the whole graph G_i is brought into main memory. The DFS-algorithm explores edges in G_i until it comes to a separator vertex again, where the whole procedure is repeated. It remains to be observed that every subgraph is entered at most $\mathcal{O}(\sqrt{B})$ times, once through each boundary vertex. Each time the graph is entered, the algorithm spends one I/O to bring it into main memory, so that $\mathcal{O}(\sqrt{B})$ I/Os are spent per subgraph. Since there are $\mathcal{O}(N/B)$ subgraphs, the I/O-bound follows.

Graphs of bounded treewidth. The treewidth of a graph has been defined by theoreticians as a parameter that captures the hardness of many NP-hard problems on this graph. In particular, many of these problems can be solved in linear time if the treewidth of the graph is constant. Recently a number of researchers have argued that the graphs produced by web crawls have constant treewidth, so that I/O-efficient algorithms for these graphs would be useful in web-modelling applications. Unfortunately the results we discuss next still are of little practical relevance because the constants hidden in the big-Oh are super-exponential in the treewidth of the graph and hence are small only for graphs of extremely small treewidth. Yet it is interesting that at least theoretically these graphs allow I/O-efficient solutions to BFS and shortest paths.

Intuitively, the treewidth of a graph G captures how far away G is from being a tree. Hence, quite naturally, the treewidth of a tree is one. A *tree-decomposition* of a graph G is a tree T storing vertices of G at its nodes. The union of these vertex sets is the vertex set of G. For every edge of G, there exists a node of T storing both endpoints of G. The nodes of T storing a vertex $v \in G$ induce a subtree of T. The width of the tree-decomposition is k if no node of T stores more than $k + 1$ vertices of G.

Under these conditions, it can be shown that the vertex set X_v stored at a node $v \in T$ is a separator that partitions G into the subgraphs defined by the subtrees of T obtained by removing v from T. Moreover, if k is constant, this separator obviously has constant size.

In [25] it is shown that if G has constant treewidth, a tree-decomposition of minimal width for G can be obtained in $\mathcal{O}(\text{sort}(N))$ I/Os. The algorithm is fairly involved and follows the internal memory algorithm by Bodlaender and Kloks [7, 8]. Given the tree-decomposition, dynamic programming can be applied to T in order to solve single source shortest paths on G. In particular, assuming that tree T is rooted at some node ρ, it is first processed from the leaves towards the root to find for every node v, the distances in $G(v)$ between all vertices in X_v, where $G(v)$ is the subgraph of G induced by all vertices stored at descendants of v. In a second phase tree T is processed from the root towards the leaves, and the information computed in the first phase is used to compute the distance from the source s to all vertices in G. Processing T bottom-up or top-down can be done using time-forward processing. Since T is a tree, this takes $\mathcal{O}(\text{scan}(N))$ I/Os.

Outerplanar graphs. A planar graph is *outerplanar* if it can be drawn in the plane so that all vertices are on the boundary of a single face. This face is called the outer face. These graphs have two properties we have seen to be useful for solving shortest paths and DFS I/O-efficiently: They are planar by definition and have treewidth at most two. Given that they are planar graphs of small treewidth, it is not surprising that for outerplanar graphs there exist extremely simple algorithms that solve shortest paths and DFS in a linear number of I/Os.

The idea behind the shortest path algorithm is to exploit the simple geometric structure of outerplanar graphs to obtain tree-decompositions of these graphs much more easily than using the general tree-decomposition algorithm. In particular, Maheshwari and Zeh [24, 32] show that an outerplanar embedding of an outerplanar graph can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. A tree-decomposition of the graph is easily obtained from the dual of the computed embedding. Once the tree-decomposition is given, the single-source shortest path problem can again be solved by applying dynamic programming to the computed tree-decomposition.

The DFS-algorithm for outerplanar graphs is based on the following observation: If the graph is biconnected, the boundary of the outer face is a simple cycle. Hence, the removal of an arbitrary edge from this cycle produces a simple path that contains all vertices of the graph and is hence a DFS-tree of the graph. If the graph is not biconnected, a DFS-tree can be obtained by "gluing" together appropriate spanning trees obtained in this manner for the biconnected components of the graph. Intuitively, the resulting tree is the same as a tree obtained by walking along the boundary of the outer face and backtracking as soon as a vertex is visited for the second time.

12.2 Planar Embedding

The planarity testing and planar embedding algorithm of [26, 32] fits very well into the line of I/O-efficient algorithms for planar graphs discussed in this course.

In particular, all the algorithms for planar graphs we have discussed use the assumption that the main memory is capable of holding planar graphs of size $\mathcal{O}(B^2)$ and then apply graph contraction ideas, sometimes somewhat disguised, to solve the problem at hand. The contraction is achieved by loading each subgraph in a B^2-partition of the graph into main memory and replacing it with another graph that encodes the relevant structural information about the boundary vertices of the graph more succinctly.

For planarity testing, all graphs $G_1, \dots, G_q$ in a B^2-partition of the given graph G are tested for planarity. If one of these graphs is non-planar, graph G cannot be planar. Otherwise each graph G_i is replaced with another planar graph G'_i of size $\mathcal{O}(B)$. Graphs $G'_1, \dots, G'_q$ are constructed so that G is planar if and only if the approximate graph A obtained as the union of graphs $G'_1, \dots, G'_q$ is planar. Since there are $\mathcal{O}(N/B^2)$ graphs $G'_1, \dots, G'_q$, and each of them has size $\mathcal{O}(B)$, graph A has size $\mathcal{O}(N/B)$.

To see that this procedure takes $\mathcal{O}(\text{sort}(N))$ I/Os, observe that each graph G_i fits into main memory. Thus, it takes $\mathcal{O}(\text{scan}(N))$ I/Os to test graphs $G_1, \dots, G_q$ for planarity and replace them with graphs $G'_1, \dots, G'_q$. Since graph A has size $\mathcal{O}(N/B)$, graph A can be tested for planarity in $\mathcal{O}(\text{scan}(N))$ I/Os using any linear-time planarity testing algorithm (e.g., [9]). The whole algorithm takes $\mathcal{O}(\text{sort}(N))$ instead of $\mathcal{O}(\text{scan}(N))$ I/Os because it takes $\mathcal{O}(\text{sort}(N))$ I/Os to compute a B^2-partition $\mathcal{P} = (S, \{G_1, \dots, G_q\})$ of G.

If graph A is reported to be planar, the algorithm of [9] also produces a planar embedding of A. Undoing the construction of graph G'_i from G_i, the embedding of each graph G'_i induced by the computed embedding of A can now be replaced with a consistent embedding of G_i. This can again be done in main memory, but requires some care.

The most difficult part of the planarity testing algorithm is to prove that graphs $G'_1, \dots, G'_q$ above exist and that each graph G'_i can be computed solely from G_i (i.e., without using any additional information about the structure of graph G.) Maheshwari and Zeh [26, 32] show that this can be done based on a decomposition of G_i into its connected, biconnected and triconnected components.

There is a subtle point about the strategy of this algorithm that is worth pointing out: It uses a planar separator algorithm to test whether the graph is planar. That is, it applies the separator algorithm without knowing whether the graph is planar. This works only because the separator algorithm from Section 9 does not use any information provided by a planar embedding of G. It is based solely on structural properties the graph is guaranteed to have if it is planar. In particular, since it is guaranteed that the separator algorithm produces a small separator in $\mathcal{O}(\text{sort}(N))$ I/Os if the graph is planar, it can be terminated with the output that G is not planar if the computed separator is too big or the algorithm starts taking too long.

12.3 Open Problems

We close with a list of interesting open problems.

Optimal separators for grid graphs. The separator for grid graphs as defined in Section 12.1 is non-optimal if the grid is sparsely populated. In particular, if all vertices in the graph are either on the separator rows or on the separator columns, the separator contains all vertices in the graph. This leads to a suboptimal performance of the shortest path algorithm. It is not hard to see that the separator algorithm for planar graphs can be modified, in order to obtain optimal separators for grid graphs; but the used machinery seems too heavy for graphs of such a simple structure. Thus, the question is whether the geometric information of grid graphs can be used to obtain optimal separators for grid graphs more easily than using the planar separator algorithm. What about weighted separators?

DFS in grid graphs and graphs of bounded treewidth. The algorithm for DFS in grid graphs discussed in Section 12.1 is non-optimal by a $\sqrt{B}$-factor. It would seem that the ideas of the DFS-algorithm for planar graphs can be adapted to obtain an optimal DFS-algorithm on grid graphs; but no positive answer has been obtained so far.

The DFS-algorithms for outerplanar and planar graphs exploit the geometry of these graphs to solve the problem in an optimal number of I/Os. The DFS-algorithm for grid graphs exploits the fact that these graphs can be partitioned into $\mathcal{O}(N/B)$ subgraphs, each of boundary size $\mathcal{O}(\sqrt{B})$. Graphs of bounded treewidth have neither a geometric structure, nor is it known how to obtain a separator partition similar to that obtainable for grid graphs. Hence, geometry-based approaches as well as approaches based on a partition into few subgraphs with small boundary size seem to fail on graphs of bounded treewidth, and the development of an I/O-efficient DFS-algorithm for this class of graphs is open.

Semi-external shortest paths. Maheshwari and Zeh [26] argue that the memory requirements of their separator algorithm can be reduced by a polylog-factor (if not to $\Theta(B)$) if the semi-external single source shortest path problem can be solved in $\mathcal{O}(\text{sort}(|E|))$ I/Os on arbitrary graphs. It is one of the most challenging open problems to determine how the assumption that the vertex set can be held in main memory can be exploited in shortest path algorithms to obtain any I/O-complexity better than $\mathcal{O}(|V| + \text{sort}(|E|))$.

Optimal connectivity. Finding the connected components of a graph is a problem that can be solved quite easily in linear time in internal memory. However, the existing I/O-efficient algorithms for this problem are by a $\log_2 \log_2(|V|B/|E|)$ factor away from optimal. While the hardness of BFS and DFS seems to stem from the fact that algorithms solving these two problems have to visit the vertices of the graph in a predetermined order (which, unfortunately, is not known to the algorithm), there is no such limiting factor for connectivity problems. The suboptimality of the existing contraction-based algorithms stems from the fact that these algorithms reduce the number of vertices by a constant factor in each iteration, but fail to achieve the same for the number of edges. An interesting question is whether there exists a smarter contraction strategy that also reduces the number of edges by a constant fraction. If such a strategy exists, optimal connectivity algorithms result. If no such strategy exists, the next thing one should look for is a search-based algorithm similar to BFS or DFS that takes advantage of the fact that the vertices can be visited in a fairly arbitrary order.

Optimal BFS, DFS, and shortest paths, or lower bounds. So far it was widely believed that an $\Omega(|V|)$ lower bound holds for the number of I/Os required to solve BFS on general graphs, while only an $\Omega(\text{perm}(|V|))$ lower bound could be shown. The BFS-algorithm of Section 6.3 disproves this conjecture. As a result, we are at a loss as to whether $\Omega\big(|V|/\sqrt{B}\big)$ is indeed a lower bound for BFS or whether BFS can be solved in $o\big(|V|/\sqrt{B}\big)$ I/Os. Any result that leads to an improvement in either direction is at the top of the wish list of most researchers working on I/O-efficient graph algorithms.

For DFS and shortest paths, we are even further away from closing the gap between the $\Omega(\text{perm}(|V|))$ lower bound and the $\mathcal{O}(|V|\log_2|V| + f(|V|,|E|))$ and $\mathcal{O}(|V| + f(|V|,|E|))$ upper bounds for these problems.

Algorithms for directed graphs. For directed graphs, no I/O-efficient shortest path algorithm is known, and the performance of the existing BFS and DFS algorithms is disappointing. Beside these results, not much is known for any problems on directed graphs. Among the most coveted are algorithms for topologically sorting directed acyclic graphs and computing the strongly connected components of a directed graph.

References

1. L. Aleksandrov and H. Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM Journal of Discrete Mathematics*, 9:129–150, 1996.
2. 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*, pages 334–345. Springer-Verlag, 1995.
3. L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proceedings of the 6th International Symposium on Algorithms and Computation*, volume 1004 of *Lecture Notes in Computer Science*, pages 82–91, 1995.
4. 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*, pages 433–447. Springer-Verlag, 2000.
5. 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*, pages 471–482. Springer-Verlag, 2001.

6. L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proceedings of the 2nd Workshop on Algorithm Engineering and Experimentation*, 2000.
7. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.
8. H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21:358–402, 1996.
9. 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*, pages 140–146, 1999.
10. G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118. Springer-Verlag, 1998.
11. 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*, pages 859–860, 2000.
12. 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*, pages 139–149, January 1995.
13. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, first edition, 1990.
14. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.
15. S. Even. *Graph Algorithms.* Computer Science Press, 1979.
16. T. Hagerup. Planar depth-first search in $O(\log n)$ parallel time. *SIAM Journal on Computing*, 19(4):678–704, August 1990.
17. F. Harary. *Graph Theory.* Addison-Wesley, 1969.
18. J. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2:135–158, 1973.
19. J. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.
20. S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.
21. 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*, pages 51–60. Springer-Verlag, July 1999. To appear in Discrete Applied Mathematics.
22. 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*, pages 169–176, October 1996.
23. R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.
24. 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*, pages 307–316. Springer-Verlag, December 1999.
25. 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*, pages 89–90, 2001.

26. 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*, pages 372–381, 2002.
27. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th Annual European Symposium on Algorithms.* Springer-Verlag, 2002. To appear.
28. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, January 1999.
29. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–159, 1972.
30. R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM Journal on Computing*, 14(4):862–874, 1985.
31. W. T. Tutte. *Graph Theory.* Cambridge University Press, 2001. First published by Addison-Wesley, 1984.
32. N. Zeh. *I/O-Efficient Algorithms for Shortest Path Related Problems.* PhD thesis, School of Computer Science, Carleton University, 2002.