4. Elementary Graph Algorithms in External Memory*

Irit Katriel and Ulrich Meyer**

4.1 Introduction

Solving real-world optimization problems frequently boils down to processing *graphs*. The graphs themselves are used to represent and structure relationships of the problem's components. In this chapter we review external-memory (**EM**) graph algorithms for a few representative problems:

Shortest path problems are among the most fundamental and also the most commonly encountered graph problems, both in themselves and as subproblems in more complex settings [21]. Besides obvious applications like preparing travel time and distance charts [337], shortest path computations are frequently needed in telecommunications and transportation industries [677], where messages or vehicles must be sent between two geographical locations as quickly or as cheaply as possible. Other examples are complex traffic flow simulations and planning tools [337], which rely on solving a large number of individual shortest path problems. One of the most commonly encountered subtypes is the *Single-Source Shortest-Path* (SSSP) version: let $G = (V, E)$ be a graph with $|V|$ nodes and $|E|$ edges, let s be a distinguished vertex of the graph, and c be a function assigning a non-negative real *weight* to each edge of G. The objective of the SSSP is to compute, for each vertex v reachable from s, the weight $\text{dist}(v)$ of a minimum-weight ("shortest") path from s to v; the weight of a path is the sum of the weights of its edges.

Breadth-First Search (BFS) [554] can be seen as the unweighted version of SSSP; it decomposes a graph into levels where level i comprises all nodes that can be reached from the source via i edges. The BFS numbers also impose an order on the nodes within the levels. BFS has been widely used since the late 1950's; for example, it is an ingredient of the classical separator algorithm for planar graphs [507].

Another basic graph-traversal approach is *Depth-First Search* (DFS) [407]; instead of exploring the graph in levels, DFS tries to visit as many graph vertices as possible in a long, deep path. When no edge to an unvisited node

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and by the DFG grant SA 933/1-1.

** Partially supported by the Center of Excellence programme of the EU under contract number ICAI-CT-2000-70025. Parts of this work were done while the author was visiting the Computer and Automation Research Institute of the Hungarian Academy of Sciences, Center of Excellence, MTA SZTAKI, Budapest.

U. Meyer et al. (Eds.): Algorithms for Memory Hierarchies, LNCS 2625, pp. 62-84, 2003.

can be found from the current node then DFS backtracks to the most recently visited node with unvisited neighbor(s) and continues there. Similar to BFS, DFS has proved to be a useful tool, especially in artificial intelligence [177]. Another well-known application of DFS is in the linear-time algorithm for finding strongly connected components [713].

Graph connectivity problems include *Connected Components* (CC), *Biconnected Components* (BCC) and *Minimum Spanning Forest* (MST/MSF). In CC we are given a graph $G = (V, E)$ and we are to find and enumerate maximal subsets of the nodes of the graph in which there is a path between every two nodes. In BCC, two nodes are in the same subset iff there are *two* edge-disjoint paths connecting them. In MST/MSF the objective is to find a spanning tree of G (spanning forest if G is not connected) with a minimum total edge weight. Both problems are central in network design; the obvious applications are checking whether a communications network is connected or designing a minimum cost network. Other applications for CC include clustering, e.g., in computational biology [386] and MST can be used to approximate the *traveling salesman* problem within a factor of 1.5 [201].

We use the standard model of external memory computation [755]: There is a main memory of size M and an external memory consisting of D disks. Data is moved in blocks of size B consecutive words. An I/O-operation can move up to D blocks, one from each disk. Further details about models for memory hierarchies can be found in Chapter 1. We will usually describe the algorithms under the assumption $D = 1$. In the final results, however, we will provide the I/O-bounds for general $D \geq 1$ as well. Furthermore, we shall frequently use the following notational shortcuts: $\mathrm{scan}(x) := \mathcal{O}(x/(D \cdot B))$, $\mathrm{sort}(x) := \mathcal{O}(x/(D \cdot B) \cdot \log_{M/B}(x/B))$, and $\mathrm{perm}(x) := \mathcal{O}(\min\{x/D, \mathrm{sort}(x)\})$.

Organization of the Chapter. We discuss external-memory algorithms for all the problems listed above. In Sections 4.2 – 4.7 we cover graph traversal problems (BFS, DFS, SSSP) and Sections 4.8 – 4.13 provide algorithms for graph connectivity problems (CC, BCC, MSF).

4.2 Graph-Traversal Problems: BFS, DFS, SSSP

In the following sections we will consider the classical graph-traversal problems Breadth-First Search (BFS), Depth-First Search (DFS), and Single-Source Shortest-Paths (SSSP). All these problems are well-understood in *internal memory* (**IM**): BFS and DFS can be solved in $\mathcal{O}(|V| + |E|)$ time [21], SSSP with nonnegative edge weights requires $\mathcal{O}(|V| \cdot \log |V| + |E|)$ time [252, 317]. On more powerful machine models, SSSP can be solved even faster [374, 724].

Most **IM** algorithms for BFS, DFS, and SSSP visit the vertices of the input graph G in a one-by-one fashion; appropriate candidate nodes for the next vertex to be visited are kept in some data-structure Q (a queue for BFS, a stack for DFS, and a priority-queue for SSSP). After a vertex v is extracted from Q, the *adjacency list* of v, i.e., the set of neighbors of v in G, is examined in order to update Q: unvisited neighboring nodes are inserted into Q; the priorities of nodes already in Q may be updated.

The Key Problems. The short description above already contains the main difficulties for I/O-efficient graph-traversal algorithms:

(a) *Unstructured indexed access to adjacency lists.*
(b)Remembering visited nodes.
(c) (The lack of) *Decrease_Key* operations in external priority-queues.

Whether (a) is problematic or not depends on the sizes of the adjacency lists; if a list contains k edges then it takes $\Theta(1 + k/B)$ I/Os to retrieve all its edges. That is fine if $k = \Omega(B)$, but wasteful if $k = \mathcal{O}(1)$. In spite of intensive research, so far there is no general solution for (a) on sparse graphs: unless the input is known to have special properties (for example *planarity*), virtually all **EM** graph-traversal algorithms require $\Theta(|V|)$ I/Os to access adjacency lists. Hence, we will mainly focus on methods to avoid spending one I/O for each edge on general graphs[1]. However, there is recent progress for BFS on arbitrary *undirected* graphs [542]; e.g., if $|E| = \mathcal{O}(|V|)$, the new algorithm requires just $\mathcal{O}(|V|/\sqrt{B} + \mathrm{sort}(|V|))$ I/Os. While this is a major step forward for BFS on undirected graphs, it is currently unclear whether similar results can be achieved for undirected DFS/SSSP or BFS/DFS/SSSP on general directed graphs.

Problem (b) can be partially overcome by solving the graph problems *in phases* [192]: a dictionary DI of maximum capacity $|\mathrm{DI}| < M$ is kept in internal memory; DI serves to remember visited nodes. Whenever the capacity of DI is exhausted, the algorithms make a pass through the external graph representation: all edges pointing to visited nodes are discarded, and the remaining edges are compacted into new adjacency lists. Then DI is emptied, and a new phase starts by visiting the next element of Q. This *phase-approach* explored in [192] is most efficient if the quotient $|V|/|\mathrm{DI}|$ is small[2]; $\mathcal{O}(\lceil |V|/|\mathrm{DI}| \rceil \cdot \mathrm{scan}(|V| + |E|))$ I/Os are needed in total to perform all graph compactions. Additionally, $\mathcal{O}(|V| + |E|)$ operations are performed on Q.

As for SSSP, problem (c) is less severe if (b) is resolved by the phase-approach: instead of actually performing Decrease_Key operations, several priorities may be kept for each node in the external priority-queue; after a node v is dequeued for the first time (with the smallest key) any further appearance of v in Q will be ignored. In order to make this work, superfluous

[1] In contrast, the chapter by Toma and Zeh in this volume (Chapter 5) reviews improved algorithms for special graph classes such as planar graphs.

[2] The chapters by Stefan Edelkamp (Chapter 11) and Rasmus Pagh (Chapter 2) in this book provide more details about space-efficient data-structures.

elements still kept in the **EM** data structure of Q are marked obsolete right before DI is emptied at the end of a phase; the marking can be done by scanning Q.

Plugging-in the I/O-bounds for external queues, stacks, and priority-queues as presented in Chapter 2 we obtain the following results:

Problem	Performance with the phase-approach [192]
BFS, DFS	$\mathcal{O}\left(\lvert V\rvert + \left\lceil \frac{\lvert V\rvert}{M} \right\rceil \cdot \mathrm{scan}(\lvert V\rvert + \lvert E\rvert)\right)$ I/Os
SSSP	$\mathcal{O}\left(\lvert V\rvert + \left\lceil \frac{\lvert V\rvert}{M} \right\rceil \cdot \mathrm{scan}(\lvert V\rvert + \lvert E\rvert) + \mathrm{sort}(\lvert E\rvert)\right)$ I/Os

Another possibility is to solve (b) and (c) by applying extra bookkeeping and extra data structures like the I/O-efficient *tournament tree* of Kumar and Schwabe [485]. In that case the graph traversal can be done in *one* phase, even if $n \gg M$.

It turns out that the best known **EM** algorithms for *undirected* graphs are simpler and/or more efficient than their respective counterparts for directed graphs; due to the new algorithm of [542], the difference for BFS on sparse graphs is currently as big as $\Omega(\sqrt{B} \cdot \log \lvert V\rvert)$.

Problems (b) and (c) usually disappear in the *semi-external memory* (**SEM**) setting where it is assumed that $M = c \cdot \lvert V\rvert < \lvert E\rvert$ for some appropriately chosen positive constant c: e.g., the **SEM** model may allow to keep a boolean array for (b) in internal memory; similarly, a node priority queue with Decrease_Key operation for (c) could reside completely in **IM**.

Still, due to (a), the currently best **EM**/**SEM** algorithms for BFS, DFS and SSSP require $\Omega(\lvert V\rvert + \lvert E\rvert / B)$ I/Os on general *directed* graphs. Taking into consideration that naive applications of the best **IM** algorithms in external memory cause $\mathcal{O}(\lvert V\rvert \cdot \log \lvert V\rvert + \lvert E\rvert)$ I/Os we see how little has been achieved so far concerning general sparse directed graphs. On the other hand, it is perfectly unclear, whether one can do significantly better at all: the best known lower-bound is only $\Omega(\mathrm{perm}(\lvert V\rvert) + \lvert E\rvert / B)$ I/Os (by the trivial reductions of list ranking to BFS, DFS, and SSSP).

In the following we will present some *one-pass approaches* in more detail. In Section 4.3 we concentrate on algorithms for undirected BFS. Section 4.4 introduces I/O-efficient Tournament Trees; their application to undirected **EM** SSSP is discussed in Section 4.5. Finally, Section 4.6 provides traversal algorithms for directed graphs.

4.3 Undirected Breadth-First Search

Our exposition of one-pass BFS algorithms for undirected graphs is structured as follows. After some preliminary observations we review the basic BFS algorithm of Munagala and Ranade [567] in Section 4.3.1. Then, in Section 4.3.2, we present the recent improvement by Mehlhorn and Meyer [542]. This algorithm can be seen as a refined implementation of the Munagala/Ranade approach. The new algorithm clearly outperforms a previous BFS approach [550], which was the first to achieve $o(|V|)$ I/Os on undirected sparse graphs with bounded node degrees. A tricky combination of both strategies might help to solve *directed* BFS with sublinear I/O; see Section 4.7 for more details.

We restrict our attention to computing the BFS *level* of each node v, i.e., the minimum number of edges needed to reach v from the source. For undirected graphs, the respective BFS tree or the BFS numbers (order of the nodes in a level) can be obtained efficiently: in [164] it is shown that each of the following transformations can be done using $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os: **BFS Numbers → BFS Tree → BFS Levels → BFS Numbers**.

The conversion BFS Numbers → BFS Tree is done as follows: for each node $v \in V$, the parent of v in the BFS tree is the node v' with BFS number $\mathit{bfsnum}(v') = \min_{(v,w)\in E} \mathit{bfsnum}(w)$. The adjacency lists can be augmented with the BFS numbers by sorting. Another scan suffices to extract the BFS tree edges.

As for the conversion BFS Tree → BFS Levels, an Euler tour [215] around the undirected BFS tree can be constructed and processed using scanning and list ranking; Euler tour edges directed towards the leaves are assigned a weight $+1$ whereas edges pointing towards the root get weight -1. A subsequent prefix-sum computation [192] on the weights of the Euler tour yields the appropriate levels.

The last transformation BFS Levels → BFS Numbers proceeds level-by-level: having computed correct numbers for level i, the order (BFS numbers) of the nodes in level $i+1$ is given as follows: each level-$(i+1)$ node v must be a child (in the BFS tree) of its adjacent level-i node with least BFS number. After sorting the nodes of level i and the edges between levels i and $i+1$, a scan provides the adjacency lists of level-$(i+1)$ nodes with the required information.

4.3.1 The Algorithm of Munagala and Ranade

We turn to the basic BFS algorithm of Munagala and Ranade [567], MR_BFS for short. It is also used as a subroutine in more recent BFS approaches [542, 550]. Furthermore, MR_BFS is applied in the deterministic CC algorithm of [567] (which we discuss in Section 4.9).

Let $L(t)$ denote the set of nodes in BFS level t, and let $|L(t)|$ be the number of nodes in $L(t)$. MR_BFS builds $L(t)$ as follows: let $A(t) := N(L(t-1))$

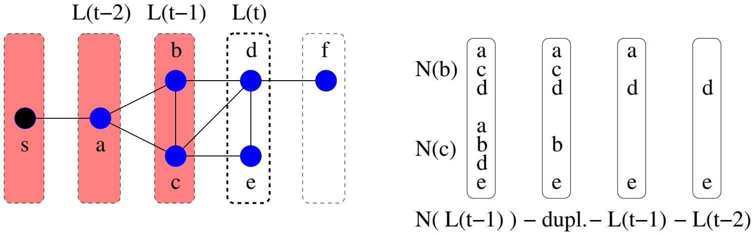

Fig. 4.1. A phase in the BFS algorithm of Munagala and Ranade [567]. Level $L(t)$ is composed out of the disjoint neighbor vertices of level $L(t-1)$ excluding those vertices already existing in either $L(t-2)$ or $L(t-1)$.

be the multi-set of neighbor vertices of nodes in $L(t-1)$; $N(L(t-1))$ is created by $|L(t-1)|$ accesses to the adjacency lists, one for each node in $L(t-1)$. Since the graph is stored in adjacency-list representation, this takes $\mathcal{O}\left(|L(t-1)| + |N(L(t-1))|/B\right)$ I/Os. Then the algorithm removes duplicates from the multi-set A. This can be done by sorting $A(t)$ according to the node indices, followed by a scan and compaction phase; hence, the duplicate elimination takes $\mathcal{O}(\mathrm{sort}(|A(t)|)$ I/Os. The resulting set $A'(t)$ is still sorted.

Now the algorithm computes $L(t) := A'(t) \setminus \{L(t-1) \cup L(t-2)\}$. Fig. 4.1 provides an example. Filtering out the nodes already contained in the sorted lists $L(t-1)$ or $L(t-2)$ is possible by parallel scanning. Therefore, this step can be done using

$$\mathcal{O}\Big(\mathrm{sort}\big(\ |N(L(t-1))|\ \big) + \mathrm{scan}\big(\ |L(t-1)| + |L(t-2)|\ \big)\Big) \text{ I/Os.}$$

Since $\sum_t |N(L(t))| = \mathcal{O}(|E|)$ and $\sum_t |L(t)| = \mathcal{O}(|V|)$, the whole execution of MR_BFS requires $\mathcal{O}(|V| + \mathrm{sort}(|E|))$ I/Os.

The correctness of this BFS algorithm crucially depends on the fact that the input graph is undirected. Assume that the levels $L(0), \ldots, L(t-1)$ have already been computed correctly. We consider a neighbor v of a node $u \in L(t-1)$: the distance from s to v is at least $t-2$ because otherwise the distance of u would be less than $t-1$. Thus $v \in L(t-2) \cup L(t-1) \cup L(t)$ and hence it is correct to assign precisely the nodes in $A'(t) \setminus \{L(t-1) \cup L(t-2)\}$ to $L(t)$.

Theorem 4.1 ([567]). *BFS on arbitrary undirected graphs can be solved using $\mathcal{O}(|V| + \mathrm{sort}(|V| + |E|))$ I/Os.*

4.3.2 An Improved BFS Algorithm

The Fast_BFS algorithm of Mehlhorn and Meyer [542] refines the approach of Munagala and Ranade [567]. It trades-off unstructured I/Os with increasing the number of iterations in which an edge may be involved. Fast_BFS operates in two phases: in a first phase it preprocesses the graph and in a second phase it performs BFS using the information gathered in the first phase. We first sketch a variant with a randomized preprocessing. Then we outline a deterministic version.

The Randomized Partitioning Phase. The preprocessing step partitions the graph into disjoint connected subgraphs $\mathcal{S}_i$, $0 \leq i \leq K$, with small expected diameter. It also partitions the adjacency lists accordingly, i.e., it constructs an external file $\mathcal{F} = \mathcal{F}_0\mathcal{F}_1 \ldots \mathcal{F}_i \ldots \mathcal{F}_{K-1}$ where $\mathcal{F}_i$ contains the adjacency lists of all nodes in $\mathcal{S}_i$. The partition is built by choosing *master nodes* independently and uniformly at random with probability $\mu = \min\{1, \sqrt{(|V| + |E|)/(B \cdot |V|)}\}$ and running a local BFS from all master nodes "in parallel" (for technical reasons, the source node s is made the master node of $\mathcal{S}_0$): in each round, each master node s_i tries to capture all unvisited neighbors of its current sub-graph $\mathcal{S}_i$; this is done by first sorting the nodes of the active fringes of all $\mathcal{S}_i$ (the nodes that have been captured in the previous round) and then scanning the dynamically shrinking adjacency-lists representation of the yet unexplored graph. If several master nodes want to include a certain node v into their partitions then an arbitrary master node among them succeeds. The selection can be done by sorting and scanning the created set of neighbor nodes.

The expected number of master nodes is $K := \mathcal{O}(1 + \mu \cdot n)$ and the expected shortest-path distance (number of edges) between any two nodes of a subgraph is at most $2/\mu$. Hence, the expected total amount of data being scanned from the adjacency-lists representation during the "parallel partition growing" is bounded by

$$X := \mathcal{O}(\sum_{v \in V} 1/\mu \cdot (1 + \text{degree}(v))) = \mathcal{O}((|V| + |E|)/\mu).$$

The total number of fringe nodes and neighbor nodes sorted and scanned during the partitioning is at most $Y := \mathcal{O}(|V| + |E|)$. Therefore, the partitioning requires

$$\mathcal{O}(\text{scan}(X) + \text{sort}(Y)) = \mathcal{O}(\text{scan}(|V| + |E|)/\mu + \text{sort}(|V| + |E|))$$

expected I/Os.

After the partitioning phase each node knows the (index of the) subgraph to which it belongs. With a constant number of sort and scan operations FAST_BFS can reorganize the adjacency lists into the format $\mathcal{F}_0\mathcal{F}_1 \ldots \mathcal{F}_i \ldots \mathcal{F}_{|\mathcal{S}|-1}$, where $\mathcal{F}_i$ contains the adjacency lists of the nodes in partition $\mathcal{S}_i$; an entry $(v, w, \mathcal{S}(w), f_{\mathcal{S}(w)})$ from the adjacency list of $v \in \mathcal{F}_i$ stands for the edge (v, w) and provides the additional information that w belongs to subgraph $\mathcal{S}(w)$ whose subfile $\mathcal{F}_{\mathcal{S}(w)}$ starts at position $f_{\mathcal{S}(w)}$ within $\mathcal{F}$. The edge entries of each $\mathcal{F}_i$ are lexicographically sorted. In total, $\mathcal{F}$ occupies $\mathcal{O}((|V| + |E|)/B)$ blocks of external storage.

The BFS Phase. In the second phase the algorithm performs BFS as described by Munagala and Ranade (Section 4.3.1) with one crucial difference: FAST_BFS maintains an external file $\mathcal{H}$ (= hot adjacency lists); it comprises unused parts of subfiles $\mathcal{F}_i$ that contain a node in the current level $L(t-1)$. FAST_BFS initializes $\mathcal{H}$ with $\mathcal{F}_0$. Thus, initially, $\mathcal{H}$ contains the adjacency list of the root node s of level $L(0)$. The nodes of each created BFS level will also carry identifiers for the subfiles $\mathcal{F}_i$ of their respective subgraphs $\mathcal{S}_i$.

When creating level $L(t)$ based on $L(t-1)$ and $L(t-2)$, FAST_BFS does not access single adjacency lists like MR_BFS does. Instead, it performs a parallel scan of the sorted lists $L(t-1)$ and $\mathcal{H}$ and extracts $N(L(t-1))$; In order to maintain the invariant that $\mathcal{H}$ contains the adjacency lists of all vertices on the current level, the subfiles $\mathcal{F}_i$ of nodes whose adjacency lists are not yet included in $\mathcal{H}$ will be merged with $\mathcal{H}$. This can be done by first sorting the respective subfiles and then merging the sorted set with $\mathcal{H}$ using one scan. Each subfile $\mathcal{F}_i$ is added to $\mathcal{H}$ at most once. After an adjacency list was copied to $\mathcal{H}$, it will be used only for $\mathcal{O}(1/\mu)$ expected steps; afterwards it can be discarded from $\mathcal{H}$. Thus, the expected total data volume for scanning $\mathcal{H}$ is $\mathcal{O}(1/\mu \cdot (|V|+|E|))$, and the expected total number of I/Os to handle $\mathcal{H}$ and $\mathcal{F}_i$ is $\mathcal{O}\left(\mu \cdot |V| + \text{sort}(|V|+|E|) + 1/\mu \cdot \text{scan}(|V|+|E|)\right)$. The final result follows with $\mu = \min\{1, \sqrt{\text{scan}(|V|+|E|)/|V|}\}$.

Theorem 4.2 ([542]). *External memory BFS on undirected graphs can be solved using* $\mathcal{O}\left(\sqrt{|V| \cdot \text{scan}(|V|+|E|)} + \text{sort}(|V|+|E|)\right)$ *expected I/Os.*

The Deterministic Variant. In order to obtain the result of Theorem 4.2 in the worst case, too, it is sufficient to modify the preprocessing phase of Section 4.3.2 as follows: instead of growing subgraphs around randomly selected master nodes, the deterministic variant extracts the subfiles $\mathcal{F}_i$ from an Euler tour [215] around a spanning tree for the connected component C_s that contains the source node s. Observe that C_s can be obtained with the deterministic connected-components algorithm of [567] using
$\mathcal{O}((1+\log\log(B \cdot |V|/|E|)) \cdot \text{sort}(|V|+|E|)) =$
$\mathcal{O}(\sqrt{|V| \cdot \text{scan}(|V|+|E|)} + \text{sort}(|V|+|E|))$ I/Os. The same number of I/Os suffices to compute a (minimum) spanning tree T_s for C_s [60].

After T_s has been built, the preprocessing constructs an Euler tour around T_s using a constant number of sort- and scan-steps [192]. Then the tour is broken at the root node s; the elements of the resulting list can be stored in consecutive order using the deterministic list ranking algorithm of [192]. This takes $\mathcal{O}(\text{sort}(|V|))$ I/Os. Subsequently, the Euler tour can be cut into pieces of size $2/\mu$ in a single scan. These Euler tour pieces account for subgraphs $\mathcal{S}_i$ with the property that the distance between any two nodes of $\mathcal{S}_i$ in G is at most $2/\mu - 1$. See Fig. 4.2 for an example. Observe that a node v of degree d may be part of $\Theta(d)$ different subgraphs $\mathcal{S}_i$. However, with a constant number of sorting steps it is possible to remove multiple node appearances and make sure that each node of C_s is part of exactly one subgraph $\mathcal{S}_i$ (actually there

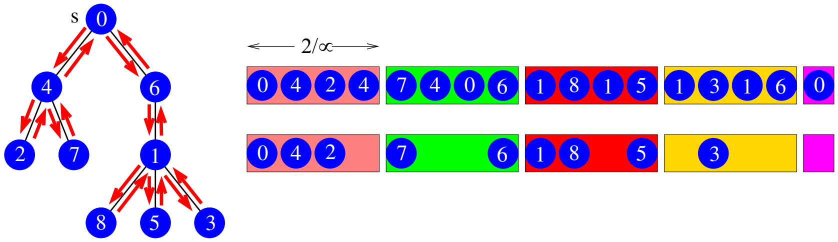

Fig. 4.2. Using an Euler tour around a spanning tree of the input graph in order to obtain a partition for the deterministic BFS algorithm.

are special algorithms for duplicate elimination, e.g. [1, 534]). Eventually, the reduced subgraphs $\mathcal{S}_i$ are used to create the reordered adjacency-list files $\mathcal{F}_i$; this is done as in the randomized preprocessing and takes another $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os. Note that the reduced subgraphs $\mathcal{S}_i$ may not be connected any more; however, this does not matter as our approach only requires that any two nodes in a subgraph are relatively close in the original input graph.

The BFS-phase of the algorithm remains unchanged; the modified preprocessing, however, guarantees that each adjacency-list will be part of the external set $\mathcal{H}$ for at most $2/\mu$ BFS levels: if a subfile $\mathcal{F}_i$ is merged with $\mathcal{H}$ for BFS level $L(t)$, then the BFS level of any node v in $\mathcal{S}_i$ is at most $L(t) + 2/\mu - 1$. Therefore, the adjacency list of v in $\mathcal{F}_i$ will be kept in $\mathcal{H}$ for at most $2/\mu$ BFS levels.

Theorem 4.3 ([542]). *External memory BFS on undirected graphs can be solved using* $\mathcal{O}\left(\sqrt{|V| \cdot \text{scan}(|V| + |E|)} + \text{sort}(|V| + |E|)\right)$ *I/Os in the worst case.*

4.4 I/O-Efficient Tournament Trees

In this section we review a data structure due to Kumar and Schwabe [485] which proved helpful in the design of better **EM** graph algorithms: the I/O-efficient tournament tree, *I/O-TT* for short. A tournament tree is a complete binary tree, where some rightmost leaves may be missing. In a figurative sense, a standard tournament tree models the outcome of a k-phase knock-out game between $|V| \leq 2^k$ players, where player i is associated with the i-th leaf of the tree; winners move up in the tree.

The I/O-TT as described in [485] is more powerful: it works as a priority queue with the Decrease_Key operation. However, both the size of the data structure and the I/O-bounds for the priority queue operations depend on the size of the universe from which the entries are drawn. Used in connection with graph algorithms, the static I/O-TT can host at most $|V|$ elements with

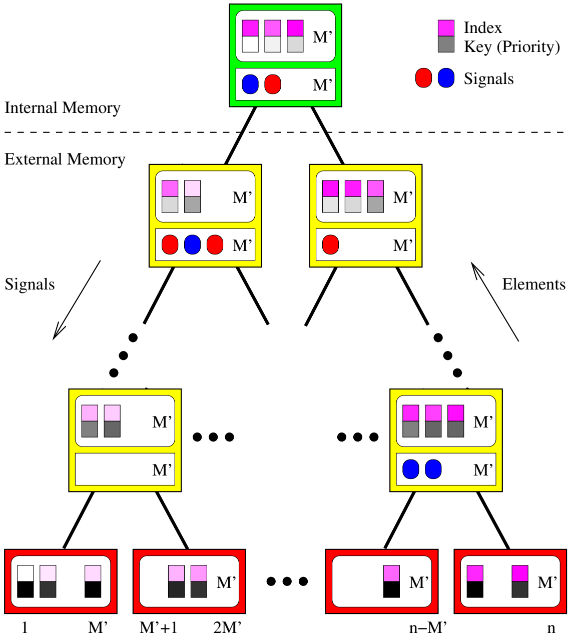

Fig. 4.3. Principle of an I/O-efficient tournament tree. Signals are traveling from the root to the leaves; elements move in opposite direction.

pairwise disjoint indices in $\{1, \ldots, |V|\}$. Besides its index x, each element also has a key k (priority). An element $\langle x_1, k_1\rangle$ is called smaller than $\langle x_2, k_2\rangle$ if $k_1 < k_2$.
The I/O-TT supports the following operations:

(i) *deletemin:* extract the element $\langle x, k\rangle$ with smallest key k and replace it by the new entry $\langle x, \infty\rangle$.
(ii) *delete(x):* replace $\langle x, \mathit{oldkey}\rangle$ by $\langle x, \infty\rangle$.
(iii) *update(x,newkey):* replace $\langle x, \mathit{oldkey}\rangle$ by $\langle x, \mathit{newkey}\rangle$ if $\mathit{newkey} < \mathit{oldkey}$.

Note that (ii) and (iii) do not require the old key to be known. This feature will help to implement the graph-traversal algorithms of Section 4.5 without paying one I/O for each edge (for example an SSSP algorithm does not have to find out explicitly whether an edge relaxation leads to an improved tentative distance).

Similar to other I/O-efficient priority queue data structures (see Chapter 2 of Rasmus Pagh for an overview) I/O-TTs rely on the concept of lazy batched processing. Let $M' = c \cdot M$ for some positive constant $c < 1$; the static I/O-TT for $|V|$ entries only has $\lceil |V|/M' \rceil$ leaves (instead of $|V|$ leaves in the standard tournament tree). Hence, there are $\mathcal{O}(\log_2(|V|/M))$ levels. Elements with indices in the range $\{(i-1) \cdot M' + 1, \ldots, i \cdot M'\}$ are mapped to the i-th leaf. The index range of internal nodes of the I/O-TT is given by the union of the index ranges of their children. Internal nodes of the I/O-TT keep a list of at least $M'/2$ and at most M' elements each (sorted according to their priorities). If the list of a tree node v contains z elements, then they are the smallest z out of all those elements in the tree being mapped to the leaves that are descendants of v. Furthermore, each internal node is equipped with a *signal buffer* of size M'. Initially, the I/O-TT stores the elements $\langle 1, +\infty\rangle, \langle 2, +\infty\rangle, \ldots, \langle |V|, +\infty\rangle$, out of which the lists of internal nodes keep at least $M'/2$ elements each. Fig. 4.3 illustrates the principle of an I/O-TT.

4.4.1 Implementation of the Operations

The operations (i)–(iii) generate *signals* which serve to propagate information down the tree; signals are inserted into the root node, which is kept in internal memory. When a signal arrives in a node it may create, delete or modify an element kept in this node; the signal itself may be discarded, altered or remain unchanged. Non-discarded signals are stored until the capacity of the node's buffer is exceeded; then they are sent down the tree towards the unique leaf node its associated element is mapped to.

Operation (i) removes an element $\langle x, k\rangle$ with smallest key k from the root node (in case there are no elements in the root it is recursively refilled from its children). A signal is sent on the path towards the leaf associated with x in order to reinsert $\langle x, +\infty\rangle$. The reinsertion takes place at the first tree node on the path with free capacity whose descendents are either empty or exclusively host elements with key infinity.

Operation (ii) is done in a similar way as (i); a delete signal is sent towards the leaf node that index x is mapped to; the signal will eventually meet the element $\langle x, \mathit{oldkey}\rangle$ and cause its deletion. Subsequently, the delete signal is converted into a signal to reinsert $\langle x, +\infty\rangle$ and proceeds as in case (i).

Finally, the signal for operation (iii) traverses its predefined tree path until either some node v_{newkey} with appropriate key range is found or the element $\langle x, \mathit{oldkey}\rangle$ is met in some node v_{oldkey}. In the latter case, if $\mathit{oldkey} > \mathit{newkey}$ then $\langle x, \mathit{newkey}\rangle$ will replace $\langle x, \mathit{oldkey}\rangle$ in the list of v_{oldkey}; if $\mathit{oldkey} \leq \mathit{newkey}$ nothing changes. Otherwise, i.e., if $\langle x, \mathit{newkey}\rangle$ belongs to a tree node v_{newkey} closer to the root than v_{oldkey}, then $\langle x, \mathit{newkey}\rangle$ will be added to the list of v_{newkey} (in case this exceeds the capacity of v_{newkey} then the largest list element is recursively flushed to the respective child node of v_{newkey} using a special flush signal). The update signal for *update(x,newkey)* is altered into a delete signal for $\langle x, \mathit{oldkey}\rangle$, which is sent down the tree in order to eliminate the obsolete entry for x with the old key.

It can be observed that each operation from (i)–(iii) causes at most two signals to travel all the way down to a leaf node. Overflowing buffers with $X > M'$ signals can be emptied using $\mathcal{O}(X/B)$ I/Os. Elements moving up the tree can be charged to signals traveling down the tree. Arguing more formally along these lines, the following amortized bound can be shown:

Theorem 4.4 ([485]). *On an I/O-efficient tournament tree with $|V|$ elements, any sequence of z delete/deletemin/update operations requires at most $\mathcal{O}(z/B \cdot \log_2(|V|/B))$ I/Os.*

4.5 Undirected SSSP with Tournament Trees

In the following we sketch how the I/O-efficient tournament tree of Section 4.4 can be used in order to obtain improved **EM** algorithms for the single source shortest path problem. The basic idea is to replace the data structure Q for the candidate nodes of **IM** traversal-algorithms (Section 4.2) by the **EM** tournament tree. The resulting SSSP algorithm works for undirected graphs with strictly positive edge weights.

The SSSP algorithm of [485] constructs an I/O-TT for the $|V|$ vertices of the graph and sets all keys to infinity. Then the key of the source node is updated to zero. Subsequently, the algorithm operates in $|V|$ iterations similarly to Dijkstra's approach [252]: iteration i first performs a *deletemin* operation in order to extract an element $\langle v_i, k_i \rangle$; the final distance of the extracted node v_i is given by $\text{dist}(v_i) = k_i$. Then the algorithm issues $\mathit{update}(w_j, \text{dist}(v_i) + c(v_i, w_j))$ operations on the I/O-TT for each adjacent edge (v_i, w_j), $v_i \neq w_j$, having weight $c(v_i, w_j)$; in case of improvements the new tentative distances will automatically materialize in the I/O-TT.

However, there is a problem with this simple approach; consider an edge (u, v) where $\text{dist}(u) < \text{dist}(v)$. By the time v is extracted from the I/O-TT, u is already settled; in particular, after removing u, the I/O-TT replaces the extracted entry $\langle u, \text{dist}(u) \rangle$ by $\langle u, +\infty \rangle$. Thus, performing $\mathit{update}(u, \text{dist}(v) + c(v, u) < \infty)$ for the edge (v, u) after the extraction of v would reinsert the settled node u into the set Q of candidate nodes. In the following we sketch how this problem can be circumvented:

A second **EM** priority-queue[3], denoted by *SPQ*, supporting a sequence of z *deletemin* and *insert* operations with (amortized) $\mathcal{O}(z/B \cdot \log_2(z/B))$ I/Os is used in order to remember settled nodes "at the right time". Initially, SPQ is empty. At the beginning of iteration i, the modified algorithm additionally checks the smallest element $\langle u'_i, k'_i \rangle$ from SPQ and compares its key k'_i with the key k_i of the smallest element $\langle u_i, k_i \rangle$ in I/O-TT. Subsequently, only the element with smaller key is extracted (in case of a tie, the element in the I/O-TT is processed first). If $k_i < k'_i$ then the algorithm proceeds as described above; however, for each $\mathit{update}(v, \text{dist}(u) + c(u, v))$ on the I/O-TT it additionally inserts $\langle u, \text{dist}(u) + c(u, v) \rangle$ into the SPQ. On the other hand, if $k'_i < k_i$ then a $\mathit{delete}(u'_i)$ operation is performed on I/O-TT as well and a new phase starts.

[3] Several priority queue data structures are appropriate; see Chapter 2 for an overview.

Operation	I/O-TT	SPQ
...	$\langle u, \text{dist}(u)\rangle$, $\langle v, *\rangle$	
$u = \mathit{TT\_deletemin()}$	$\langle v, *\rangle$	
$\mathit{TT\_update}(v, \text{dist}(u) + c(u,v))$	$\langle v, *'\rangle$	
$\mathit{SPQ\_insert}(u, \text{dist}(u) + c(u,v))$	$\langle v, *'\rangle$	$\langle u, \text{dist}(u) + c(u,v)\rangle$
...	$\langle v, \text{dist}(v)\rangle$	$\langle u, \text{dist}(u) + c(u,v)\rangle$
$v = \mathit{TT\_deletemin()}$		$\langle u, \text{dist}(u) + c(u,v)\rangle$
$\mathit{TT\_update}(u, \text{dist}(v) + c(u,v))$	$\langle u, \text{dist}(v) + c(u,v)\rangle$	$\langle u, \text{dist}(u) + c(u,v)\rangle$
...	$\langle u, \text{dist}(v) + c(u,v)\rangle$	$\langle u, \text{dist}(u) + c(u,v)\rangle$
$u = \mathit{SPQ\_deletemin()}$	$\langle u, \text{dist}(v) + c(u,v)\rangle$	
$\mathit{TT\_delete}(u)$		

Fig. 4.4. Identifying spurious entries in the I/O-TT with the help of a second priority queue SPQ.

In Fig. 4.4 we demonstrate the effect for the previously stated problem concerning an edge (u, v) with $\text{dist}(u) < \text{dist}(v)$: after node u is extracted from the I/O-TT for the first time, $\langle u, \text{dist}(u) + c(u, v)\rangle$ is inserted into SPQ. Since $\text{dist}(u) < \text{dist}(v) \leq \text{dist}(u) + c(u, v)$, node v will be extracted from I/O-TT while u is still in SPQ. The extraction of v triggers a spurious reinsertion of u into I/O-TT having key $\text{dist}(v) + c(v, u) = \text{dist}(v) + c(u, v) > \text{dist}(u) + c(u, v)$. Thus, u is extracted as the smallest element in SPQ before the reinserted node u becomes the smallest element in I/O-TT; as a consequence, the resulting $\mathit{delete}(u)$ operation for I/O-TT eliminates the spurious node u in I/O-TT just in time. Extra rules apply for nodes with identical shortest path distances.

As already indicated in Section 4.2, one-pass algorithms like the one just presented still require $\Theta(|V| + (|V| + |E|)/B)$ I/Os for accessing the adjacency lists. However, the remaining operations are more I/O-efficient: $\mathcal{O}(|E|)$ operations on the I/O-TT and SPQ add another $\mathcal{O}(|E|/B \cdot \log_2(|E|/B))$ I/Os. Altogether this amounts to $\mathcal{O}(|V| + |E|/B \cdot \log_2(|E|/B))$ I/Os.

Theorem 4.5. *SSSP on undirected graphs can be solved using* $\mathcal{O}(|V| + |E|/B \cdot \log_2(|E|/B))$ *I/Os.*

The unpublished full version of [485] also provides a one-pass **EM** algorithm for DFS on undirected graphs. It requires $\mathcal{O}((|V| + |E|/B) \cdot \log_2 |V|)$ I/Os. A different algorithm for directed graphs achieving the same bound will be sketched in the next section.

4.6 Graph-Traversal in Directed Graphs

The best known one-pass traversal-algorithms for general directed graphs are often less efficient and less appealing than their undirected counterparts from the previous sections. The key difference is that it becomes much more complicated to keep track of previously visited nodes of the graph; the nice trick of checking a constant number of previous levels for visited nodes as discussed for undirected BFS does not work for directed graphs. Therefore we store edges that point to previously seen nodes in a so-called *buffered repository tree* (BRT) [164]: A BRT maintains $|E|$ elements with keys in $\{1, \ldots, |V|\}$ and supports the operations $insert(edge, key)$ and $extract\_all(key)$; the latter operation reports and deletes all edges in the BRT that are associated with the specified key.

A BRT can be built as a height-balanced static binary tree with $|V|$ leaves and buffers of size B for each internal tree node; leaf i is associated with graph node v_i and stores up to degree(v_i) edges. Insertions into the BRT happen at the root; in case of buffer overflow an inserted element (e, i) is flushed down towards the i-th leaf. Thus, an insert operation requires amortized $\mathcal{O}(1/B \cdot \log_2 |V|)$ I/Os. If $extract\_all(i)$ reports x edges then it needs to read $\mathcal{O}(\log_2 |V|)$ buffers on the path from the root to the i-th leaf; another $\mathcal{O}(x/B)$ disk blocks may have to be read at the leaf itself. This accounts for $\mathcal{O}(x/B + \log_2 |V|)$ I/Os.

For DFS, an external stack S is used to store the vertices on the path from the root node of the DFS tree to the currently visited vertex. A step of the DFS algorithm checks the previously unexplored outgoing edges of the topmost vertex u from S. If the target node v of such an edge (u, v) has not been visited before then u is the father of v in the DFS tree. In that case, v is pushed on the stack and the search continues for v. Otherwise, i.e., if v has already been visited before, the next unexplored outgoing edge of u will be checked. Once all outgoing edges of the topmost node u on the stack have been checked, node u is popped and the algorithm continues with the new topmost node on the stack.

Using the BRT the DFS procedure above can be implemented I/O efficiently as follows: when a node v is encountered for the first time, then for each incoming edge $e_i = (u_i, v)$ the algorithm performs $insert(e_i, u_i)$. If at some later point u_i is visited then $extract\_all(u_i)$ provides a list of all edges out of u_i that should not be traversed again (since they lead to nodes already seen before). If the (ordered) adjacency list of u_i is kept in some **EM** priority-queue $P(u_i)$ then all superfluous edges can be deleted from $P(u_i)$ in an I/O-efficient way. Subsequently, the next edge to follow is given by extracting the minimum element from $P(u_i)$.

The algorithm takes $\mathcal{O}(|V| + |E|/B)$ I/Os to access adjacency lists. There are $\mathcal{O}(|E|)$ operations on the n priority queues $P(\cdot)$ (implemented as external buffer trees). As the DFS algorithm performs an inorder traversal of a DFS tree, it needs to change between different $P(\cdot)$ at most $\mathcal{O}(|V|)$ times. Therefore, $\mathcal{O}(|V| + \text{sort}(|E|))$ I/Os suffice to handle the operations on all $P(\cdot)$. Additionally, there are $\mathcal{O}(|E|)$ *insert* and $\mathcal{O}(|V|)$ *extract_all* operations on the BRT; the I/Os required for them add up to $\mathcal{O}((|V| + |E|/B) \cdot \log_2 |V|)$ I/Os.

The algorithm for BFS works similarly, except that the stack is replaced by an external queue.

Theorem 4.6 ([164, 485]). *BFS and DFS on directed graphs can be solved using $\mathcal{O}((|V| + |E|/B) \cdot \log_2 |V|)$ I/Os.*

4.7 Conclusions and Open Problems for Graph Traversal

In the previous sections we presented the currently best **EM** traversal algorithms for general graphs assuming a single disk. With small modifications, the results of Table 4.1 can be obtained for $D \geq 1$ disks.

Table 4.1. Graph traversal with D parallel disks.

Problem	I/O-Bound
Undir. BFS	$\mathcal{O}\left(\sqrt{\|V\| \cdot \text{scan}(\|V\| + \|E\|)} + \text{sort}(\|V\| + \|E\|)\right)$
Dir. BFS, DFS	$\mathcal{O}\left(\min\left\{\|V\| + \left\lceil\frac{\|V\|}{M}\right\rceil \cdot \text{scan}(\|V\| + \|E\|),\ \left(\|V\| + \frac{\|E\|}{D \cdot B}\right) \cdot \log_2 \|V\|\right\}\right)$
Undir. SSSP	$\mathcal{O}\left(\min\left\{\|V\| + \left\lceil\frac{\|V\|}{M}\right\rceil \cdot \text{sort}(\|V\| + \|E\|),\ \|V\| + \frac{\|E\|}{D \cdot B} \cdot \log_2 \|V\|\right\}\right)$
Dir. SSSP	$\mathcal{O}\left(\|V\| + \left\lceil\frac{\|V\|}{M}\right\rceil \cdot \text{sort}(\|V\| + \|E\|)\right)$

One of the central goals in **EM** graph-traversal is the reduction of unstructured I/O for accessing adjacency-lists. The FAST_BFS algorithm of Section 4.3.2 provides a first solution for the case of undirected BFS. However, it also raises new questions: For example, is $\Omega(|V|/\sqrt{D \cdot B})$ I/Os a lower bound for sparse graphs? Can similar results be obtained for DFS or SSSP with arbitrary nonnegative edge weights? In the following we shortly discuss difficulties concerning possible extensions towards *directed* BFS.

The improved I/O-bound of FAST_BFS stems from partitioning the undirected input graph G into disjoint node sets $\mathcal{S}_i$ such that the distance in G between any two nodes of $\mathcal{S}_i$ is relatively small. For directed graphs, a partitioning of that kind may not always exist. It could be beneficial to have a larger number of overlapping node sets where the algorithm accesses just a small fraction of them. Similar ideas underlie a previous BFS algorithm for undirected graphs with small node degrees [550]. However, there are deep problems concerning space blow-up and efficiently identifying the "right" partitioning for arbitrary directed graphs. Besides all that, an $o(|V|)$-I/O algorithm for directed BFS must also feature novel strategies to remember previously visited nodes. Maybe, for the time being, this additional complication should be left aside by restricting attention to the semi-external case; first results for semi-external BFS on directed Eulerian graphs are given in [542].

4.8 Graph Connectivity: Undirected CC, BCC, and MSF

The *Connected Components*(*CC*) problem is to create, for a graph $G = (V, E)$, a list of the nodes of the graph, sorted by component, with a special record marking the end of each component. The Connected Components Labeling (*CCL*) problem is to create a vector L of size $|V|$ such that for every $i, j \in \{1, \cdots, |V|\}$, $L[i] = L[j]$ iff nodes i and j are in the same component of G. A solution to CCL can be converted into a CC output in $\mathcal{O}(\text{sort}(|V|))$ I/Os, by sorting the nodes according to their component label and adding the separators.

Minimum Spanning Forest (*MSF*) is the problem of finding a subgraph F of the input graph G such that every connected component in G is connected in F and the total weight of the edges of F is minimal.

Biconnected Components (*BCC*) is the problem of finding subsets of the nodes such that u and v are in the same subset iff there are two node-disjoint paths between u and v.

CC and MSF are related, as an MSF yields the connected components of the graph. Another common point is that typical algorithms for both problems involve reading the adjacency lists of nodes. Reading a node's adjacency list L takes $\mathcal{O}(\text{scan}(|L|))$ I/Os, so going over the nodes in an arbitrary order and reading each node's adjacency list once requires a total of $\mathcal{O}(\text{scan}(|E|) + |V|)$ I/Os. If $|V| \leq |E|/B$, the scan$(|E|)$ term dominates.

For both CC and MST, we will see algorithms that have this $|V|$ term in their complexity. Before using such an algorithm, a *node reduction* step will be applied to reduce the number of nodes to at most $|E|/B$. Then, the $|V|$ term is dominated by the other terms in the complexity. A node reduction step should have the property that the transformations it performs on the graph preserve the solutions to the problem in question. It should also be possible to reintegrate the nodes or edges that were removed into the solution of the problem for the reduced graph. For both CC and MST, the node reduction step will apply a small number of phases of an iterative algorithm for the problem. We could repeat such phases until completion, but that would require $\Theta(\log\log|V|)$ phases, each of which is quite expensive. The combination of a small number, $\mathcal{O}(\log\log(|V|B/|E|))$, of phases with an efficient algorithm for dense graphs, gives a better complexity. Note that when $|V| \leq |E|$, $\mathcal{O}(\text{sort}(|E|)\log\log(|V|B/|E|)) = \mathcal{O}(\text{sort}(|E|)\log\log B)$.

For BCC, we show an algorithm that transforms the graph and then applies CC. BCC is clearly not easier than CC; given an input $G = (E, V)$ to CC, we can construct a graph G' by adding a new node and connecting it with each of the nodes of G. Each biconnected component of G' is the union of a connected component of G with the new node.

4.9 Connected Components

The undirected BFS algorithm of Section 4.3.1 can trivially be modified to solve the CCL problem: whenever $L(t-1)$ is empty, the algorithm has spanned a complete component and selects some unvisited node for $L(t)$. It can therefore number the components, and label each node in $L(t)$ with the current component number before $L(t)$ is discarded. This adds another $\mathcal{O}(|V|)$ I/Os, so the complexity is still $\mathcal{O}(|V| + \mathrm{sort}(|V| + |E|))$. For a dense graph, where $|V| \leq |E|/B$, we have $\mathcal{O}(|V| + \mathrm{sort}(|V| + |E|)) = \mathcal{O}(\mathrm{sort}(|V| + |E|))$. For a general graph, Munagala and Ranade [567] suggest to precede the BFS run with the following node reduction step:

The idea is to find, in each phase, sets of nodes such that the nodes in each set belong to the same connected component. Among each set a leader is selected, all edges between nodes of the set are contracted and the set is reduced to its leader. The process is then repeated on the reduced graph.

Algorithm 1. Repeat until $|V| \leq |E|/B$:

1. For each node, select a neighbor. Assuming that each node has a unique integer node-id, let this be the neighbor with the smallest id. This partitions the nodes into *pseudotrees* (directed graphs where each node has outdegree 1).
2. Select a leader from each pseudotree.
3. Replace each edge (u, v) in E by an edge $(R(u), R(v))$, where $R(v)$ is the leader of the pseudotree v belongs to.
4. Remove isolated nodes, parallel edges and self loops.

For Step 1 we sort two copies of the edges, one by source node and one by target node, and scan both lists simultaneously to find the lowest numbered neighbor of each node.

For step 2 we note that a pseudotree is a tree with one additional edge. Since each node selected its smallest neighbor, the smallest node in each pseudotree must be on the pseudotree's single cycle. In addition, this node can be identified by the fact that the edge selected for it goes to a node with a higher ID. Hence, we can scan the list of edges of the pseudoforest, remove each edge for which the source is smaller than the target, and obtain a forest while implicitly selecting the node with the smallest id of each pseudotree as the leader. On a forest, Step 3 can be implemented by pointer doubling as in the algorithm for list ranking in Chapter 3 with $\mathcal{O}(\mathrm{sort}(|E|))$ I/Os. Step 4

requires an additional constant number of sorts and scans of the edges. The total I/Os for one iteration is then $\mathcal{O}(\text{sort}(|E|))$. Since each iteration at least halves the number of nodes, $\log_2(|V|B/|E|)$ iterations are enough, for a total of $\mathcal{O}(\text{sort}(|E|)\log(|V|B/|E|))$ I/Os.
After $\log_2(|V|B/|E|)$ iterations, we have a contracted graph in which each node represents a set of nodes from the original graph. Applying BFS on the contracted graph gives a component label to each supernode. We then need to go over the nodes of the original graph and assign to each of them the label of the supernode it was contracted to. This can be done by sorting the list of nodes by the id of the supernode that the node was contracted to and the list of component labels by supernode id, and then scanning both lists simultaneously.
The complexity can be further improved by contracting more edges per phase at the same cost. More precisely, in phase i up to $\sqrt{S_i}$ edges adjacent to each node will be contracted, where $S_i = 2^{(3/2)^i} \left(= S_{i-1}^{3/2}\right)$ (less edges will be contracted only if some nodes have become singletons, in which case they become inactive). This means that the number of active nodes at the beginning of phase i, $|V_i|$, is at most $|V|/S_{i-1} \cdot S_{i-2} \leq |V|/(S_i)^{2/3}(S_i)^{4/9} \leq |V|/S_i$, and $\log\log(|V|B/|E|)$ phases are sufficient to reduce the number of nodes as desired.

To stay within the same complexity per phase, phase i is executed on a reduced graph G_i, which contains only the relevant edges: those that will be contracted in the current phase. Then $|E_i| \leq |V_i|\sqrt{S_i}$. We will later see how this helps, but first we describe the algorithm:

Algorithm 2. Phase i:

1. For each active node, select up to $d = \sqrt{S_i}$ adjacent edges (less if the node's degree is smaller). Generate a graph $G_i = (V_i, E_i)$ over the active nodes with the selected edges.
2. Apply $\log d$ phases of Algorithm 1 to G_i.
3. Replace each edge (u, v) in E by an edge $(R(u), R(v))$ and remove redundant edges and nodes as in Algorithm 1.

Complexity Analysis. Steps 1 an 3 take $\mathcal{O}(\text{sort}(|E|))$ I/Os as in Algorithm 1. In Step 2, each phase of Algorithm 1 takes $\mathcal{O}(\text{sort}(|E_i|))$ I/Os. With $|E_i| \leq |V_i|\sqrt{S_i}$ (due to Step 1) and $|V_i| \leq (|V|/S_i)$ (as shown above), this is $\mathcal{O}\big(\text{sort}(|V|/\sqrt{S_i})\big)$ and all $\log\sqrt{S_i}$ phases need a total of $\mathcal{O}(\text{sort}(|V|))$ I/Os. Hence, one phase of Algorithm 2 needs $\mathcal{O}(\text{sort}(|E|))$ I/Os as before, giving a total complexity of $\mathcal{O}(\text{sort}(|E|)\log\log(|V|B/|E|))$ I/Os for the node reduction. The BFS-based CC algorithm can then be executed in $\mathcal{O}(\text{sort}(|E|))$ I/Os.

For $D > 1$, we perform node reduction phases until $|V| \leq |E|/BD$, giving:

Theorem 4.7 ([567]). *CC can be solved using*

$$\mathcal{O}(\text{sort}(|E|) \cdot \max\{1, \log\log(|V|BD/|E|)\}) \text{ I/Os.}$$

4.10 Minimum Spanning Forest

An $\mathcal{O}(|V| + \mathrm{sort}(|E|))$ MSF algorithm. The Jarník-Prim [428, 615] MSF algorithm builds the MSF incrementally, one tree at a time. It starts with an arbitrary node and in each iteration, finds the node that is connected to the current tree by the lightest edge, and adds it to the tree. When there do not exist any more edges connecting the current tree with unvisited nodes, it means that a connected component of the graph has been spanned. The algorithm then selects an arbitrary unvisited node and repeats the process to find the next tree of the forest. For this purpose, the nodes which are not in the MSF are kept in a priority queue, where the priority of a node is the weight of the lightest edge that connects it to the current MSF, and ∞ if such an edge does not exist. Whenever a node v is added to the MSF, the algorithm looks at v's neighbors. For each neighbor u which is in the queue, the algorithm compares the weight of the edge (u, v) with the priority of u. If the priority is higher than the weight of the new edge that connects u to the tree, u's priority is updated. In **EM**, this algorithm requires at least one I/O per edge to check the priority of a node, hence the number of I/Os $\Theta(|E|)$.

However, Arge et al. [60] pointed out that if *edges* are kept in the queue instead of nodes, the need for updating priorities is eliminated. During the execution of the algorithm, the queue contains (at least) all edges that connect nodes in the current MSF with nodes outside of it. It can also contain edges between nodes that are in the MSF. The algorithm proceeds as follows: repeatedly perform *extract minimum* to extract the minimum weight edge (u, v) from the queue. If v is already in the MSF, the edge is discarded. Otherwise, v is included in the MSF and all edges incident to it, except for (u, v), are inserted into the queue. Since each node in the tree inserts all its adjacent edges, if v is in the MSF, the queue contains two copies of the edge (u, v). Assuming that all edge weights are distinct, after the first copy was extracted from the queue, the second copy is the new minimum. Therefore, the algorithm can use one more *extract-minimum* operation to know whether the edge should be discarded or not.

The algorithm reads each node's adjacency list once, with $\mathcal{O}(|V| + |E|/B)$ I/Os. It also performs $\mathcal{O}(|E|)$ *insert* and *extract minimum* operations on the queue. By using an external memory priority queue that supports these operations in $\mathcal{O}\Big(1/B \log_{M/B}(N/B)\Big)$ I/Os amortized, we get that the total complexity is $\mathcal{O}(|V| + \mathrm{sort}(|E|))$.

Node Reduction for MSF. As in the CC case, we wish to reduce the number of nodes to at most $|E|/B$. The idea is to contract edges that are in the MSF, merging the adjacent nodes into supernodes. The result is a reduced graph G' such that the MSF of the original graph is the union of the contracted edges and the edges of the MSF of G', which can be found recursively.

A *Boruvka* [143] step selects for each node the minimum-weight edge incident to it. It contracts all the selected edges, replacing each connected component they define by a supernode that represents the component, removing all isolated nodes, self-edges, and all but the lowest weight edge among each set of multiple edges.

Each Boruvka step reduces the number of nodes by at least a factor of two, while contracting only edges that belong to the MSF. After i steps, each supernode represents a set of at least 2^i original nodes, hence the number of supernodes is at most $|V|/2^i$. In order to reduce the number of nodes to $|E|/B$, $\lceil \log(|V|B/|E|) \rceil$ phases are necessary. Since one phase requires $\mathcal{O}(\text{sort}(|E|))$ I/Os, this algorithm has complexity of $\mathcal{O}(\text{sort}(|E|) \cdot \max\{1, \log(|V|B/|E|))\})$.

As with the CC node reduction algorithm, this can be improved by combining phases into superphases, where each superphase still needs $\mathcal{O}(\text{sort}(|E|))$ I/Os, and reduces more nodes than the basic step. Each superphase is the same, except that the edges selected for E_i are not the smallest numbered $\sqrt{S_i}$ edges adjacent to each node, but the *lightest* $\sqrt{S_i}$ edges. Then a superphase which is equivalent to $\log \sqrt{s}_i$ *Boruvka* steps is executed with $\mathcal{O}(\text{sort}(|E|))$ I/Os. The total number of I/Os for node reduction is $\mathcal{O}(\text{sort}(|E|) \cdot \max\{1, \log\log(|V|B/|E|)\})$. The output is the union of the edges that were contracted in the node reduction phase and the MSF of the reduced graph.

Theorem 4.8 ([567]). *MSF can be solved using*

$$\mathcal{O}(\text{sort}(|E|) \cdot \max\{1, \log\log(|V|BD/|E|)\}) \text{ I/Os.}$$

4.11 Randomized *CC* and *MSF*

We now turn to the randomized MSF algorithm that was proposed by Abello, Buchsbaum and Westbrook [1], which is based on an internal memory algorithm that was found by Karger, Klein and Tarjan [445]. As noted above, an MSF algorithm is also a CC algorithm.

Let $G = (V, E)$ be the input graph with edge weights $c(e)$. Let $E' \subseteq E$ be an arbitrary subset of the edges of G and let F' be the MSF of $G' = (V, E')$. Denote by $W_{F'}(u, v)$ the weight of the heaviest edge on the path connecting u and v in F'. If such a path does not exist, $W_{F'}(u, v) = \infty$.

Lemma 4.9. *For any edge $e = (u, v) \in E$, if $w(e) \geq W_{F'}(u, v)$, then there exists an MSF that does not include e.*

The lemma follows immediately from the *cycle property*: the heaviest edge of each cycle is not in the MSF. □

The following MSF algorithm is based on Lemma 4.9 [445]:

1. Apply two Boruvka phases to reduce the number of nodes by at least a factor of 4. Call the contracted graph G'.

2. Choose a subgraph H of G' by selecting each edge independently with probability p.
3. Apply the algorithm recursively to find the MSF F of H.
4. Delete from G' each edge $e = (u, v)$ for which $w(e) > W_{F'}(u, v)$. Call the resulting graph G''.
5. Apply the algorithm recursively to find the MSF F' of G''.
6. return the union of the edges contracted in step 1 and the edges of F'.

Step 1 requires $\mathcal{O}(\text{sort}(|E|))$ I/Os and Step 2 an additional $\mathcal{O}(\text{scan}(|E|))$ I/Os.
Step 4 can be done with $\mathcal{O}(\text{sort}(|E|))$ I/Os as well [192]. The idea is based on the MST verification algorithm of King [456], which in turn uses a simplification of Komlós's algorithm for finding the heaviest weight edge on a path in a tree [464]. Since Komlós's algorithm only works on full branching trees[4], the $MSF\ F'$ is first converted into a forest of full branching trees F'' such that $W_{F'}(u, v) = W_{F''}(u, v)$ and F'' has $\mathcal{O}(|V|)$ nodes. The conversion is done as follows: each node of F' is a leaf in F''. A sequence of Boruvka steps is applied to F' and each step defines one level of each tree in F'': the nodes at distance i from the leaves are the supernodes that exist after i Boruvka steps and the parent of each non-root node is the supernode into which is was contracted. Since contraction involves at least two nodes, the degree of each non-leaf node is at least 2. All Boruvka steps require a total of $\mathcal{O}(\text{sort}(|V|))$ I/Os.

The next step is to calculate, for each edge (u, v) in G', the least common ancestor (LCA) of u and v in F''. Chiang et al. show that this can be done with $\mathcal{O}(\text{sort}(|E|))$ I/Os [192]. An additional $\mathcal{O}(\text{sort}(|E|))$ I/Os are necessary to construct a list of tuples, one for each edge of G', of the weight of the edge, its endpoints and the LCA of the endpoints. These tuples are then filtered through F'' with $\mathcal{O}((|E|/|V|)\text{sort}(|V|)) = \mathcal{O}(\text{sort}(|E|))$ I/Os as follows: each tuple is sent as a query to the root of the tree and traverses down from it towards the leaves that represent the endpoints of the edge. When the query reaches the LCA, it is split into two weight-queries, one continuing towards each of the endpoints. If a weight query traverses a tree edge which is heavier than the query edge, the edge is not discarded. Otherwise, it is. To make this I/O efficient, queries are passed along tree paths using batch filtering [345]. This means that instead of traversing the tree from root to leaves for each query, a batch of $|V|$ queries is handled at each node before moving on to the next.

The non-recursive stages then require $\mathcal{O}(\text{sort}(|E|))$ I/Os. Karger et al., have shown that the expected number of edges in G'' is $|V|/p$ [445]. The algorithm includes two recursive calls. One with at most $|V|/4$ nodes and expected $p|E|$ edges, and the other with at most $|V|/4$ nodes and expected $|V|/p$ edges. The total expected I/O complexity is then:

[4] A full branching tree is a tree in which all leaves are at the same level and each non-leaf node has at least two descendants

$$t(|E|,|V|) \le \mathcal{O}(\mathrm{sort}(|E|)) + t\left(p|E|, \frac{|V|}{4}\right) + t\left(\frac{|V|}{p}, \frac{|V|}{4}\right) = \mathcal{O}(\mathrm{sort}(|E|))$$

Theorem 4.10 ([1]). *MSF and CC can be solved by a randomized algorithm that uses $\mathcal{O}(\mathrm{sort}(|E|))$ I/Os in the expected case.*

4.12 Biconnected Components

Tarjan and Vishkin [714] propose a parallel algorithm that reduces *BCC* to an instance of *CC*. The idea is to transform the graph G into a graph G' such that the connected components of G' correspond to the biconnected components of G. Each node of G' represents an edge of G. If two edges e_1 and e_2 are on the same simple cycle in G, the edge (e_1, e_2) exists in G'. The problem with this construction is that the graph G' is very large; it has $|E|$ nodes and up to $\Omega\left(|E|^2\right)$ edges. However, they show how to generate a smaller graph G'' with the desired properties: instead of including all edges of G', first find (any) rooted spanning tree T of G and generate G'' as the the subgraph of G' induced by the edges of T. Formally, we say that two nodes of T are *unrelated* if neither is a predecessor of the other. G'' then includes the edges:

1. $\{(u,v),(x,w)\}$ such that $(u,v),(x,w) \in T$, $(v,w) \in G - T$ and v, w are unrelated in T.
2. $\{(u,v),(v,w)\}$ such that $(u,v),(v,w) \in T$ and there exists an edge in G between a descendant of w and a non-descendant of v.

It is not difficult to see that the connected components of G'' correspond to biconnected components of G, and that the number of edges in G'' is $O(|E|)$.

Chiang et al. [192] adapt this algorithm to the I/O model. Finding an arbitrary spanning tree is obviously not more difficult than finding an MST, which can be done with $\mathcal{O}(\mathrm{sort}(|E|) \cdot \max\{1, \log\log(|V|BD/|E|)\})$ I/Os (Theorem 4.8). To construct G'', we can use Euler tour and list ranking (Chapter 3) to number the nodes by preorder and find the number of descendants of each node, followed by a constant number of sorts and scans of the edges to check the conditions described above. Hence, the construction of G'' after we have found the MSF needs $\mathcal{O}(\mathrm{sort}(|V|))$ I/Os. Finding the connected components of G'' has the same complexity as MSF (Theorem 4.7). Deriving the biconnected components of G from the connected components of G'' can then be done with a constant number of sorts and scans of the edges of G and G''.

Theorem 4.11 ([192]). *BCC can be solved using*

$$\mathcal{O}(\mathrm{sort}(|E|) \cdot \max\{1, \log\log(|V|BD/|E|)\}) \text{ I/Os.}$$

Using the result of Theorem 4.10 we get:

Theorem 4.12. *BCC can be solved by a randomized algorithm using $\mathcal{O}(\mathrm{sort}(|E|))$ I/Os.*

4.13 Conclusion for Graph Connectivity

Munagala and Ranade [567] prove a lower bound of $\Omega\left(|E|/|V| \cdot \mathrm{sort}(|V|)\right)$ I/Os for CC, BCC and MSF. Note that $|E|/|V| \cdot \mathrm{sort}(|V|) = \Theta\left(\mathrm{sort}(|E|)\right)$.

We have surveyed randomized algorithms that achieve this bound, but the best known deterministic algorithms have a slightly higher I/O complexity. Therefore, while both deterministic and randomized algorithms are efficient, there still exists a gap between the upper bound and the lower bound in the deterministic case.

Acknowledgements We would like to thank the participants of the GI-Dagstuhl Forschungsseminar “Algorithms for Memory Hierarchies” for a number of fruitful discussions and helpful comments on this chapter.