External-Memory Breadth-First Search with Sublinear I/O[⋆]

Kurt Mehlhorn and Ulrich Meyer

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany.

Abstract. Breadth-first search (BFS) is a basic graph exploration technique. We give the first external memory algorithm for sparse undirected graphs with sublinear I/O. The best previous algorithm requires $\Theta(n + \frac{n+m}{D \cdot B} \cdot \log_{M/B} \frac{n+m}{B})$ I/Os on a graph with n nodes and m edges and a machine with main-memory of size M, D parallel disks, and block size B. We present a new approach which requires only $\mathcal{O}(\sqrt{\frac{n \cdot (n+m)}{D \cdot B}} + \frac{n+m}{D \cdot B} \cdot \log_{M/B} \frac{n+m}{B})$ I/Os. Hence, for $m = \mathcal{O}(n)$ and all realistic values of $\log_{M/B} \frac{n+m}{B}$, it improves upon the I/O-performance of the best previous algorithm by a factor $\Omega(\sqrt{D \cdot B})$. Our approach is fairly simple and we conjecture it to be practical. We also give improved algorithms for undirected single-source shortest-paths with small integer edge weights and for semi-external BFS on directed Eulerian graphs.

1 Introduction

Breadth-First Search (**BFS**) is a basic graph-traversal method. It decomposes the input graph G of n nodes and m edges into at most n levels where level i comprises all nodes that can be reached from a designated source s via a path of i edges. BFS is used as a subroutine in many graph algorithms; the paradigm of breadth-first search also underlies shortest-paths computations. In this paper we focus on BFS for general undirected graphs and sparse directed Eulerian graphs; Eulerian graphs contain a cycle that traverses every edge of the graph precisely once.

External-memory (**EM**) computation is concerned with problems that are too large to fit into main memory (internal memory, **IM**). The central issue of **EM** computation is that accessing the secondary memory takes several orders of magnitude longer than performing an internal operation. We use the standard model of **EM** computation [16]. 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. For graphs with n nodes and m edges the semi-external memory (**SEM**) setting assumes $c \cdot n \leq M < m$ for some appropriate constant $c \geq 1$.

A number of basic computational problems can be solved I/O-efficiently. The most prominent example is **EM** sorting [2, 15]: sorting x items of constant size takes $\mathrm{sort}(x) =$

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

$\Theta(\frac{x}{D \cdot B} \cdot \log_{M/B} \frac{x}{B})$ I/Os. BFS, however, seems to be hard for external-memory computation (and also parallel computation). Even the best **SEM** BFS algorithms known require $\Omega(n)$ I/Os.

Recall the standard $\mathcal{O}(n + m)$-time internal-memory BFS algorithm. It visits 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 a FIFO queue Q. 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. Running this algorithm in external memory will result in $\Theta(n + m)$ I/Os. In this bound the $\Theta(n)$-term results from the unstructured accesses to the adjacency lists; the $\Theta(m)$-term is caused by m unstructured queries to find out whether neighboring nodes have already been unvisited.

The best **EM** BFS algorithm known (Munagala and Ranade [14]) overcomes the latter problem; it requires $\Theta(n + \text{sort}(n + m))$ I/Os on general undirected graphs. However, the Munagala/Ranade algorithm still pays one I/O for each node.

In this paper we show how to overcome the first problem as well: the new algorithm for undirected graphs needs just $\mathcal{O}(\sqrt{\frac{n \cdot (n+m)}{D \cdot B}} + \text{sort}(n + m))$ I/Os. Our approach is simple and has a chance to be practical. We also discuss extensions to undirected single-source shortest-paths (SSSP) with small integer edge weights and semi-external BFS on directed Eulerian graphs.

This paper is organized as follows. In Section 2 we review previous work and put our work into context. In Section 3 we outline a randomized version of our new approach. The details are the subject of Section 5. We start with a review of the algorithm of Munagala and Ranade (Section 4) and then discuss our improvement (Sections 5.1 and 5.2). Section 6 presents a deterministic version of our new approach. In Section 7 we sketch an extension to some single-source shortest-paths problem. Another modification yields an improved semi-external BFS algorithm for sparse directed Eulerian graphs (Section 8). Finally, Section 9 provides some concluding remarks and open problems.

2 Previous Work and New Results

Previous Work. I/O-efficient algorithms for graph-traversal have been considered in, e.g., [1, 3, 4, 7–14]. In the following we will only discuss results related to BFS.

The currently fastest BFS algorithm for general undirected graphs [14] requires $\Theta(n + \text{sort}(m))$ I/Os. The best bound known for directed **EM** BFS is $\mathcal{O}(\min\{n + \frac{n}{M} \cdot \frac{m}{D \cdot B}, (n + \frac{m}{D \cdot B}) \cdot \log_2 \frac{n}{D \cdot B}\})$ I/Os [7–9]. This also yields an $\mathcal{O}(n + \frac{m}{D \cdot B})$-I/O algorithm for **SEM** BFS.

Faster algorithms are only known for special types of graphs: $\mathcal{O}(\text{sort}(n))$ I/Os are sufficient to solve **EM** BFS on trees [7], grid graphs [5], outer-planar graphs [10], and graphs of bounded tree width [11]. Slightly sublinear I/O was known for undirected graphs with bounded maximum node degree d: the algorithm [13] needs $\mathcal{O}(\frac{n}{\gamma \cdot \log_d(D \cdot B)} + \text{sort}(n \cdot (D \cdot B)^\gamma))$ I/Os and $\mathcal{O}(n \cdot (D \cdot B)^\gamma)$ external space for an arbitrary parameter $0 < \gamma \leq 1/2$. Maheshwari and Zeh [12] proposed I/O-optimal algorithms for a number of fundamental problems on planar graphs; in particular, they show how to compute BFS on planar graphs using $\mathcal{O}(\text{sort}(n))$ I/Os.

SSSP can be seen as the weighted version of BFS. Consequently, all known **EM** SSSP algorithms do not perform better than the respective **EM** BFS algorithms listed above. The best known lower bound for BFS is $\Omega(\min\{n, \mathrm{sort}(n)\} + \frac{n+m}{D \cdot B})$ I/Os. It follows from the respective lower bound for the list-ranking problem [8].

New Results. We present a new **EM** BFS algorithm for undirected graphs. It comes in two versions (randomized and deterministic) and requires only $\mathcal{O}(\sqrt{\frac{n \cdot (n+m)}{D \cdot B}} + \mathrm{sort}(n+m))$ I/Os (expected or worst-case, respectively). For sparse graphs with $m = \mathcal{O}(n)$ and realistic machine parameters, the $\mathcal{O}(\sqrt{\frac{n \cdot (n+m)}{D \cdot B}})$-term in the I/O-bound will be dominant. In that case our approach improves upon the I/O-performance of the best previous algorithm [14] by a factor of $\Omega(\sqrt{D \cdot B})$. More generally, the new algorithm is asymptotically superior to the old algorithm for $m = o(\frac{D \cdot B \cdot n}{\log_{M/B} n/B})$; on denser graphs both approaches require $\mathcal{O}(\mathrm{sort}(n+m))$ I/Os.

A simple extension of our new BFS algorithm solves the SSSP problem on undirected graphs with integer edge-weights in $\{1, \ldots, W\}$ for small W: it requires $\mathcal{O}(\sqrt{\frac{W \cdot n \cdot (n+m)}{D \cdot B}} + W \cdot \mathrm{sort}(n+m))$ I/Os. After another modification we obtain an improved algorithm for **SEM** BFS on sparse directed Eulerian graphs: it achieves $\mathcal{O}(\frac{n+m}{(D \cdot B)^{1/3}} + \mathrm{sort}(n+m) \cdot \log n)$ I/Os. A performance comparison for our BFS algorithms is depicted in Figure 1.

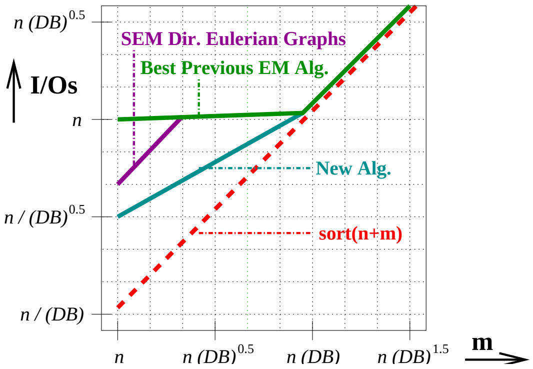

Fig. 1. Comparison: I/O-performance of the new BFS algorithms.

3 High-Level Description of the New BFS Algorithm

Our algorithm refines the algorithm of Munagala and Ranade [14] which constructs the BFS tree level-by-level. It operates in two phases. In a first phase it preprocesses the graph and in the second phase it performs BFS using the information gathered in the first phase.

The preprocessing partitions the graph into disjoint subgraphs $\mathcal{S}_i$, $0 \le i \le K$ with small average internal shortest-path distances. 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 randomized partition is created by choosing seed nodes independently and uniformly at random with probability μ and running a BFS starting from all seed nodes. Then the expected number of seed nodes is $K = \mathcal{O}(\mu \cdot n)$ and the expected shortest-path distance between any two nodes of a subgraph is at most $\mathcal{O}(1/\mu)$. The expected I/O-bound for constructing the partition is $\mathcal{O}(\frac{n+m}{\mu \cdot D \cdot B} + \mathrm{sort}(n+m))$.

In the second phase we perform BFS as described by Munagala and Ranade with one crucial difference. We maintain an external file $\mathcal{H}$ (= hot adjacency lists) which is essentially the union of all $\mathcal{F}_i$ such that the current level of the BFS-tree contains a node in $\mathcal{S}_i$. Thus it suffices to scan $\mathcal{H}$ (i.e., to access the disk blocks of $\mathcal{H}$ in consecutive fashion) in order to construct the next level of the tree. Each subfile $\mathcal{F}_i$ is added to $\mathcal{H}$ at most once; this involves at most $\mathcal{O}(K + \mathrm{sort}(n+m))$ I/Os in total. We prove that after an adjacency list was copied to $\mathcal{H}$, it will be used only for $\mathcal{O}(1/\mu)$ steps on the average; afterwards the respective list can be discarded from $\mathcal{H}$. We obtain a bound of $\mathcal{O}(\mu \cdot n + \frac{n+m}{\mu \cdot D \cdot B} + \mathrm{sort}(n+m))$ on the expected number of I/Os for the second phase. Choosing $\mu = \min\{1, \sqrt{\frac{n+m}{n \cdot D \cdot B}}\}$ gives our bound.

4 The Algorithm of Munagala and Ranade

We review the BFS algorithm of Munagala and Ranade [14], MR_BFS for short. We restrict 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 can be obtained efficiently: in [7] it is shown that each of the following transformations can be done using $\mathcal{O}(\mathrm{sort}(n+m))$ I/Os: BFS Numbers → BFS Tree → BFS Levels → BFS Numbers.

Let $L(t)$ denote the set of nodes in BFS level t, and let $A(t) := N(L(t-1))$ be the multi-set of neighbors of nodes in $L(t-1)$. MR_BFS builds $L(t)$ as follows: $A(t)$ is created by $|L(t-1)|$ accesses to the adjacency lists of all nodes in $L(t-1)$. This causes $\mathcal{O}(|L(t-1)| + |A(t)|/(D \cdot B))$ I/Os. Observe that $\mathcal{O}(1 + x/(DB))$ I/Os are needed to read a list of length x. Then the algorithm removes duplicates from $A(t)$. 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 MR_BFS computes $L(t) := A'(t) \backslash \{L(t-1) \cup L(t-2)\}$. Figure 2 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 $\mathcal{O}((|A(t)| + |L(t-1)| + |L(t-2)|)/(D \cdot B))$ I/Os. Since $\sum_t |A(t)| = \mathcal{O}(m)$ and $\sum_t |L(t)| = \mathcal{O}(n)$, MR_BFS requires $\mathcal{O}(n + \mathrm{sort}(n+m))$ I/Os. The $\Theta(n)$ I/Os result from the unstructured accesses to the n adjacency lists.

The correctness of this BFS algorithm crucially depends on the input graph being undirected: assume inductively that levels $L(0), \ldots, L(t-1)$ have already been computed correctly and consider a neighbor v of a node $u \in L(t-1)$. Then the distance

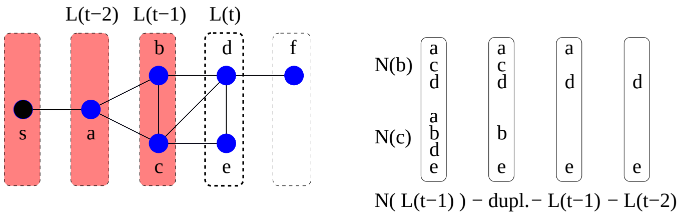

Fig. 2. A phase in the BFS algorithms of Munagala and Ranade [14]. Level $L(t)$ is composed out of the disjoint neighbor vertices of level $L(t-1)$ excluding those vertices already existing in either $L(t-2)$ or $L(t-1)$.

from the source node 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 1 ([14]). *Undirected BFS can be solved using $\mathcal{O}(n + \text{sort}(n+m))$ I/Os.*

5 The New Approach

We show how to speed-up the Munagala/Ranade approach (MR_BFS) of the previous section. We refer to the resulting algorithm as FAST_BFS. We may assume w.l.o.g. that the input graph is connected (otherwise we may run the randomized $\mathcal{O}(\text{sort}(n+m))$-I/O connected-components algorithm of [1] and only keep the nodes and edges of the component C_s that contains the source node; all nodes outside of C_s will be assigned BFS-level infinity, and the BFS computation continues with C_s). We begin with the randomized preprocessing part of FAST_BFS:

5.1 Partitioning a Graph into Small Distance Subgraphs

As a first step, FAST_BFS restructures the adjacency lists of the graph representation: it grows disjoint connected subgraphs $\mathcal{S}_i$ from randomly selected nodes s_i and stores the adjacency lists of the nodes in $\mathcal{S}_i$ in an external file $\mathcal{F}$. The node s_i is called the *master node* of subgraph S_i. A node is selected to be a master with probability $\mu = \min\{1, \sqrt{\frac{n+m}{n \cdot D \cdot B}}\}$. This choice of μ minimizes the total cost of the algorithm as we will see later. Additionally, we make sure that the source node s of the graph will be the master of partition $\mathcal{S}_0$. Let K be the number of master nodes. Then $\mathrm{E}[K] = 1 + \mu n$.

The partitioning is generated "in parallel": in each round, each master node s_i tries to capture all unvisited neighbors of its current sub-graph $\mathcal{S}_i$. If several master nodes want to include a certain node v into their partitions then an arbitrary master node among them succeeds.

At the beginning of a phase, the adjacency lists of the nodes lying on the boundaries of the current partitions are active; they carry the label of their master node. A scan through the set of adjacency lists removes these labeled lists, appends them in no particular order to the file $\mathcal{F}$, and forms a set of requests for the involved target nodes. Subsequently, the requests are sorted. A parallel scan of the sorted requests and the *shrunken* representation for the unvisited parts of the graph allows us to identify and label the new boundary nodes (and their adjacency lists). Each adjacency list is active during at most one phase. The partitioning procedure stops once there are no active adjacency lists left.

The expected I/O-performance of the preprocessing step depends on the speed with which the graph representation shrinks during the partitioning process.

Lemma 1. *Let $v \in G$ be an arbitrary node; then v is assigned to some subgraph (and hence is removed from the graph representation) after an expected number of at most $1/\mu$ rounds.*

Proof: Consider a shortest path $\mathcal{P} = \langle s, u_j, \ldots, u_2, u_1, v\rangle$ from the source node s to v in G. Let k, $1 \leq k \leq j$, be the smallest index such that u_k is a master node. Then v is assigned to a subgraph in or before the k-th round. Due to the random selection of master nodes, we have $\mathbf{E}[k] \leq 1/\mu$. □

The parallel randomized construction of the partitions also implies:

Corollary 1. *Consider an arbitrary node $v \in \mathcal{S}_i$ and let s_i be the master node of the subgraph $\mathcal{S}_i$. The expected shortest-path distance between v and s_i in S_i is at most $1/\mu$.*

By Lemma 1, the expected total amount of data being processed during the partitioning is bounded by $X := \mathcal{O}(\sum_{v\in V} 1/\mu \cdot (1 + \mathrm{degree}(v))) = \mathcal{O}((n+m)/\mu)$. However, sorting only occurs for active adjacency lists. Thus the preprocessing requires $\mathcal{O}((n+m)/(\mu \cdot D \cdot B) + \mathrm{sort}(n+m))$ 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 we can partition 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 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 $\mathcal{O}((n+m)/B)$ blocks of external storage (spread over the D disks in round-robin fashion). $\mathcal{F}$ consists of K subfiles with $\mathrm{E}[K] = 1 + \mu \cdot n$. The size of the subfiles may vary widely. Some spread out over several disk blocks and some may share the same disk block. The following lemma summarizes the discussion.

Lemma 2. *The randomized preprocessing of* FAST_BFS *requires $\mathcal{O}(\frac{n+m}{\mu \cdot D \cdot B} + \mathrm{sort}(n+m))$ expected I/Os.*

5.2 The BFS Phase

We construct the BFS levels one by one as in the algorithm of Munagala and Ranade (MR_BFS). The novel feature of our algorithm is the use of a sorted external file $\mathcal{H}$. We initialize $\mathcal{H}$ with $\mathcal{F}_0$. Thus, in particular, $\mathcal{H}$ contains the adjacency list of the source 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 $\mathcal{H}$. While doing so, it extracts the adjacency lists of all nodes $v_j \in L(t-1)$ that can be found in $\mathcal{H}$. Let $V_1 \subseteq L(t-1)$ be the set of nodes whose adjacency lists could be obtained in that way. In a second step, FAST_BFS extracts from $L(t-1)$ the partition identifiers of those nodes in $V_2 := L(t-1) \setminus V_1$. After sorting these identifiers and eliminating duplicates, FAST_BFS knows which subfiles $\mathcal{F}_i$ of $\mathcal{F}$ contain the missing adjacency lists. The respective subfiles are concatenated into a temporary file $\mathcal{F}'$ and then sorted. Afterwards the missing adjacency lists for the nodes in V_2 can be extracted with a simple scan-step from the sorted $\mathcal{F}'$ and the remaining adjacency lists can be merged with the sorted set $\mathcal{H}$ in one pass.

After the adjacency lists of the nodes in $L(t-1)$ have been obtained, the set $N(L(t-1))$ of neighbor nodes can be generated with a simple scan. At this point the augmented format of the adjacency lists is used in order to attach the partition information to each node in $N(L(t-1))$. Subsequently, FAST_BFS proceeds like MR_BFS: it removes duplicates from $N(L(t-1))$ and also discards those nodes that have already been assigned to BFS levels $L(t-1)$ and $L(t-2)$. The remaining nodes constitute $L(t)$. The constructed levels are written to external memory as a consecutive stream of data, thus occupying $\mathcal{O}(n/(D \cdot B))$ blocks striped over the D disks.

Since FAST_BFS is simply a refined implementation of MR_BFS, correctness is preserved. We only have to reconsider the I/O-bounds:

Lemma 3. *The BFS-phase of* FAST_BFS *requires* $\mathcal{O}(\mu \cdot n + \frac{n+m}{\mu \cdot D \cdot B} + \mathrm{sort}(n+m))$ *expected I/0s.*

Proof: Apart from the preprocessing of FAST_BFS (Lemma 2) we mainly have to deal with the amount of I/Os needed to maintain the data structure $\mathcal{H}$. For the construction of BFS level $L(t)$, the contents of the sorted sets $\mathcal{H}$, $L(t-2)$, and $L(t-1)$ will be scanned a constant number of times. The first $D \cdot B$ blocks of $\mathcal{H}$, $L(t-2)$, and $L(t-1)$ are always kept in main memory. Hence, scanning these data items does not necessarily cause I/O for each level. External memory access is only needed if the data volume is $\Omega(D \cdot B)$. In that case, however, the number of I/Os needed to scan x data items over the whole execution of FAST_BFS is bounded by $\mathcal{O}(x/(D \cdot B))$.

Unstructured I/O happens when $\mathcal{H}$ is filled by merging subfiles $\mathcal{F}_i$ with the current contents of $\mathcal{H}$. For a certain BFS level, data from several subfiles $\mathcal{F}_i$ may be added to $\mathcal{H}$. However, the data of each single $\mathcal{F}_i$ will be merged with $\mathcal{H}$ at most once. Hence, the number of I/Os needed to perform the mergings can be split between

(a) the adjacency lists being loaded from $\mathcal{F}$ and
(b) those already being in $\mathcal{H}$.

The I/O bound for part (a) is $\mathcal{O}(\sum_i (1 + \frac{|\mathcal{F}_i|}{D \cdot B} \cdot \log_{M/B} \frac{n+m}{B})) = \mathcal{O}(K + \mathrm{sort}(n+m))$ I/Os, and $\mathbf{E}[K] = 1 + \mu \cdot n$.

With respect to (b) we observe first that the adjacency list A_v of an arbitrary node $v \in \mathcal{S}_i$ stays in $\mathcal{H}$ for an expected number at most of $2/\mu$ rounds. This follows from the fact that the expected shortest-path distance between any two nodes of a subgraph is at most $2/\mu$: let $L(t')$ be the BFS level for which $\mathcal{F}_i$ (and hence A_v) was merged with $\mathcal{H}$. Consequently, there must be a node $v' \in \mathcal{S}_i$ that belongs to BFS level $L(t')$. Let s_i be the master node of subgraph $\mathcal{S}_i$ and let $d(x, y)$ denote the number of edges on the shortest path between the nodes x and y in $\mathcal{S}_i$. Since the graph is undirected, the BFS level of v will lie between $L(t')$ and $L(t' + d(v', s_i) + d(s_i, v))$. As soon as v becomes assigned to a BFS level, A_v is discarded from $\mathcal{H}$. By Corollary 1, $\mathbf{E}[d(v', s_i) + d(s_i, v)] \leq 2/\mu$. In other words, each adjacency list is part of $\mathcal{H}$ for expected $\mathcal{O}(2/\mu)$ BFS-levels. Thus, the expected total data volume for (b) is bounded by $\mathcal{O}((n + m)/\mu)$. This results in $\mathcal{O}((n+m)/(\mu \cdot D \cdot B))$ expected I/Os for scanning $\mathcal{H}$ during merge operations. By the same argumentation, each adjacency list in $\mathcal{H}$ takes part in at most $\mathcal{O}(1/\mu)$ scan-steps for the generation of $N(L(\cdot))$ and $L(\cdot)$. Similar to MR_BFS, scanning and sorting all BFS levels and sets $N(L(\cdot))$ takes $\mathcal{O}(\mathrm{sort}(n + m))$ I/Os. □

Combining Lemmas 2 and 3 and making the right choice of μ yields:

Theorem 2. *External memory BFS on arbitrary undirected graphs can be solved using $\mathcal{O}(\sqrt{\frac{n\cdot(n+m)}{D\cdot B}} + \mathrm{sort}(n + m))$ expected I/Os.*

Proof: By our lemmas the expected number of I/0s is bounded by $\mathcal{O}(\mu \cdot n + \frac{n+m}{\mu\cdot D\cdot B} + \mathrm{sort}(n + m))$. The expression is minimized for $\mu^2 \cdot n \cdot D \cdot B = n + m$. Choosing $\mu = \min\{1, \sqrt{n \cdot D \cdot B/(n + m)}\}$ the stated bound follows. □

6 The Deterministic Variant

In order to obtain the result of Theorem 2 in the worst case, too, it is sufficient to modify the preprocessing phase of Section 5.1 as follows: instead of growing subgraphs around randomly selected master nodes, the deterministic variant extracts the subfiles $\mathcal{F}_i$ from an Euler Tour around the 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 [14] using $\mathcal{O}((1 + \log\log(D \cdot B \cdot n/m)) \cdot \mathrm{sort}(n + m))$ I/Os. The same amount of I/O suffices to compute a (minimum) spanning tree T_s for C_s [3].

After T_s has been built, the preprocessing constructs an Euler Tour around T_s using a constant number of sort- and scan-steps [8]. Then the tour is broken at the source node s; the elements of the resulting list can be stored in consecutive order using the deterministic list-ranking algorithm of [8]. This causes $\mathcal{O}(\mathrm{sort}(n))$ I/Os. Subsequently, the Euler Tour can be chopped into pieces of size $1/\mu$ with a simple scan step. 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 $1/\mu - 1$. 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 duplicates and hence make sure that each node of C_s is part of exactly one subgraph $\mathcal{S}_i$, for example of the one with the smallest index; in particular, $s \in \mathcal{S}_0$. Eventually, the reduced subgraphs $\mathcal{S}_i$ are used to create the re-ordered adjacency-list files $\mathcal{F}_i$; this is done as in the old preprocessing and takes another $\mathcal{O}(\mathrm{sort}(n + m))$ I/Os.

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 $1/\mu$ 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 $L(t) + 1/\mu - 1$. The bound on the total number of I/Os follows from the fact that $\mathcal{O}((1 + \log\log(D \cdot B \cdot n/m)) \cdot \mathrm{sort}(n+m)) = \mathcal{O}(\sqrt{\frac{n\cdot(n+m)}{D\cdot B}} + \mathrm{sort}(n+m))$.

Theorem 3. *There is a deterministic algorithm that solves external memory BFS on undirected graphs using* $\mathcal{O}(\sqrt{\frac{n\cdot(n+m)}{D\cdot B}} + \mathrm{sort}(n+m))$ *I/Os.*

7 Extension to some SSSP Problem

We sketch how to modify FAST_BFS in order to solve the Single-Source Shortest-Paths (SSSP) problem on undirected graphs with integer edge-weights in $\{1, \ldots, W\}$ for small W. Due to the "BFS-bottleneck" all previous algorithms for SSSP required $\Omega(n)$ I/Os. Our simple extension of FAST_BFS needs $\mathcal{O}(\sqrt{\frac{W\cdot n\cdot(n+m)}{D\cdot B}} + W \cdot \mathrm{sort}(n+m))$ I/Os. Thus, for sparse graphs and constant W the resulting algorithm FAST_SSSP requires $\mathcal{O}(\frac{n}{\sqrt{D\cdot B}} + \mathrm{sort}(n))$ I/Os.

For integer weights in $\{1, \ldots, W\}$, the maximum shortest-path distance of an arbitrary reachable node from the source node s is bounded by $W \cdot (n-1)$. FAST_SSSP subsequently identifies the set of nodes with shortest-path distances $1, 2, \ldots$, denoted by levels $L(1)$, $L(2)$, …; for $W > 1$ some levels will be empty. During the construction of level $L(t)$, FAST_SSSP keeps the first $D \cdot B$ blocks of each level $L(t-W-1), \ldots, L(t+W-1)$ in main memory. The neighbor nodes $N(L(t-1))$ of $L(t-1)$ are put to $L(t), \ldots, L(t+W-1)$ according to the edge weights. After discarding duplicates from the tentative set $L(t)$, it is checked against $L(t-W-1), \ldots, L(t-1)$ in order to remove previously labeled nodes from $L(t)$. For the whole algorithm this causes at most $\mathcal{O}(W \cdot \mathrm{sort}(n+m))$ I/Os.

Using the randomized preprocessing of FAST_BFS, the expected length of stay for an arbitrary adjacency list in $\mathcal{H}$ is multiplied by a factor of at most W (as compared to FAST_BFS): the expected shortest-path distance between any two nodes u, $v \in \mathcal{S}_i$ is at most $W \cdot \mathbf{E}[d(u, s_i) + d(s_i, v)] \leq 2 \cdot W/\mu$. Hence, the expected number of I/Os to handle $\mathcal{H}$ is at most $\mathcal{O}(W/\mu \cdot (n+m)/(D \cdot B) + W \cdot \mathrm{sort}(n+m))$. The choice $\mu = \min\{1, \sqrt{\frac{W\cdot(n+m)}{n\cdot D\cdot B}}\}$ balances the costs of the various phases and the stated bound results.

8 Semi-External BFS on Sparse Directed Eulerian Graphs

The analysis for FAST_BFS as presented in the previous sections does not transfer to *directed* graphs. There are two main reasons: (i) In order to detect previously labeled nodes during the construction of BFS level $L(t)$ it is usually not sufficient to check just $L(t-1)$ and $L(t-2)$; a node in $A'(t)$ may already have appeared before in any level $L(0), \ldots, L(t-1)$. (ii) Unless a subgraph $\mathcal{S}_i$ is strongly connected it is not guaranteed that once a node $v \in \mathcal{S}_i$ is found to be part of BFS level $L(t)$, all other nodes $v' \in \mathcal{S}_i$ will belong to BFS some levels $L(t')$ having $t' < t + |\mathcal{S}_i|$; in other words, adjacency lists for nodes in $\mathcal{S}_i$ may stay (too) long in the data structure $\mathcal{H}$.

Problem (i) can be circumvented in the **SEM** setting by keeping a lookup table in internal memory. Unfortunately, we do not have a general solution for (ii). Still we obtain an improved I/O-bound for **SEM** BFS on sparse directed *Eulerian graphs*[1]. The preprocessing is quite similar to the deterministic undirected variant of Section 6. However, instead of grouping the adjacency lists based on an Euler Tour around a *spanning tree*, we partition them concerning an Euler Circuit for the *whole graph*:

The PRAM algorithm of [6] yields an Euler Circuit in $\mathcal{O}(\log n)$ time; it applies $\mathcal{O}(n+m)$ processors and uses $\mathcal{O}(n+m)$ space. Hence, this parallel algorithm can be converted into an **EM** algorithm which requires $\mathcal{O}(\log n \cdot \mathrm{sort}(n+m))$ I/Os [8]. Let $\langle v_0, v_1, \ldots, v_{m-1}, v_m \rangle$ denote the order of the nodes on an Euler Circuit for G, starting from one occurrence of the source node, i.e., $v_0 = s$. Let the subgraph $\mathcal{S}_i$ contain the nodes of the multi-set $\{v_{i \cdot (D \cdot B)^{1/3}}, \ldots, v_{(i+1) \cdot (D \cdot B)^{1/3} - 1}\}$. As in the deterministic preprocessing for the undirected case (Section 6), a node v may be part of several subgraphs $\mathcal{S}_i$; therefore, v's adjacency list will only be kept in exactly one subfile $\mathcal{F}_i$. We impose another additional restriction: the subfiles $\mathcal{F}_i$ only store adjacency lists of nodes having outdegree at most $(D \cdot B)^{1/3}$; these nodes will be called *light*, nodes with outdegree larger than $(D \cdot B)^{1/3}$ are called *heavy*. The adjacency lists for heavy nodes are kept in a standard representation for adjacency lists.

The BFS-phase of the directed **SEM** version differs from the fully-external undirected approach in two aspects: (i) The BFS level $L(t)$ is constructed as $A'(t) \setminus \{L(0) \cup L(1) \cup \ldots \cup L(t-1)\}$, where $L(0), L(1), \ldots, L(t-1)$ are kept in internal memory. (ii) The adjacency list of each *heavy* node v is accessed separately using $\mathcal{O}(1 + \mathit{outdegree}(v)/(D \cdot B))$ I/Os at the time the lists needs to be read. Adjacency lists of heavy nodes are *not* inserted into the data structure $\mathcal{H}$. Each such adjacency list will be accessed at most once. As there are at most $m/(D \cdot B)^{1/3}$ heavy nodes this accounts for $\mathcal{O}(m/(D \cdot B)^{1/3})$ extra I/Os.

Theorem 4. *Semi-external memory BFS on directed Eulerian graphs requires* $\mathcal{O}(\frac{n+m}{(D \cdot B)^{1/3}} + \mathrm{sort}(n+m) \cdot \log n)$ *I/Os in the worst case.*

Proof: As already discussed before the modified preprocessing can be done using $\mathcal{O}(\mathrm{sort}(n+m) \cdot \log n)$ I/Os.

The amount of data kept in each $\mathcal{F}_i$ is bounded by $\mathcal{O}((D \cdot B)^{2/3})$. Hence, accessing and merging all the $m/(D \cdot B)^{1/3}$ subfiles $\mathcal{F}_i$ into $\mathcal{H}$ during the BFS-phase takes $\mathcal{O}(m/(D \cdot B)^{1/3} + \mathrm{sort}(m))$ I/Os (excluding I/Os to scan data already stored in $\mathcal{H}$).

A subfile $\mathcal{F}_i$ is called *regular* if none of its adjacency lists stays in $\mathcal{H}$ for more than $2 \cdot (D \cdot B)^{1/3}$ successive BFS levels; otherwise, $\mathcal{F}_i$ is called *delayed*. The total amount of data kept and scanned in $\mathcal{H}$ from regular subfiles is at most $\mathcal{O}(m/(D \cdot B)^{1/3} \cdot (D \cdot B)^{2/3} \cdot (D \cdot B)^{1/3}) = \mathcal{O}(m \cdot (D \cdot B)^{2/3})$. This causes $\mathcal{O}(m/(D \cdot B)^{1/3})$ I/Os.

[1] An Euler Circuit of a graph is a cycle that traverses every edge of the graph precisely once. A graph containing an Euler Circuit is called Eulerian. If a directed graph is connected then it is Eulerian provided that, for every vertex v, $\mathit{indegree}(v) = \mathit{outdegree}(v)$.

Now we turn to the delayed subfiles: let $\mathcal{D} := \{\mathcal{F}_{i_0}, \mathcal{F}_{i_1}, \ldots, \mathcal{F}_{i_k}\}$, $k \leq m/(D \cdot B)^{1/3}$, be the set of all delayed subfiles, where $i_j < i_{j+1}$. Furthermore, let t_{i_j} be the time (BFS level) when $\mathcal{F}_{i_j}$ is loaded into $\mathcal{H}$; similarly let t'_{i_j} be the time (BFS level) after which all data from $\mathcal{F}_{i_j}$ has been removed from $\mathcal{H}$ again.

Recall that the source node s is the first node on the Euler Circuit $\langle v_0, v_1, \ldots, v_{m-1}, v_m, v_0 \rangle$. Hence, node v_i has BFS level at most i. Furthermore, if v_i belongs to BFS level $x \leq i$ then the successive node v_{i+1} on the Euler Circuit has BFS level at most $x + 1$. As $\mathcal{F}_{i_0}$ contains (a subset of) the adjacency lists of the light nodes in the multi-set $\{v_{i_0 \cdot (D \cdot B)^{1/3}}, \ldots, v_{(i_0+1) \cdot (D \cdot B)^{1/3} - 1}\}$, we find $t'_{i_0} \leq (i_0 + 1) \cdot (D \cdot B)^{1/3}$. More generally,

$$t'_{i_j} \leq t_{i_{j-1}} + (i_j - i_{j-1} + 1) \cdot (D \cdot B)^{1/3}.$$

The formula captures the following observation: once all data of $\mathcal{F}_i$ has been loaded into $\mathcal{H}$, the data of $\mathcal{F}_{i+l}$ will have been completely processed after the next $(l+1) \cdot (D \cdot B)^{1/3}$ BFS levels the latest. As each $\mathcal{F}_i$ contains at most $\mathcal{O}((D \cdot B)^{2/3})$ data, the total amount of data scanned in $\mathcal{H}$ from delayed $\mathcal{F}_i$ is bounded by $Z = \sum_{j=0}^{k} (t'_{i_j} - t_{i_j}) \cdot (D \cdot B)^{2/3}$ $\leq (i_0 + 1) \cdot (D \cdot B) + \sum_{j=1}^{k} (t_{i_{j-1}} - t_{i_j} + (i_j - i_{j-1} + 1) \cdot (D \cdot B)^{1/3}) \cdot (D \cdot B)^{2/3}$. The latter sum telescopes, and Z is easily seen to be bounded by $t_k \cdot (D \cdot B)^{2/3} + (i_k + k + 1) \cdot (D \cdot B)$. Together with $k, i_k \leq m/(D \cdot B)^{1/3}$ and $t_k \leq n$ this implies another $\mathcal{O}((n + m)/(D \cdot B)^{1/3})$ I/Os. □

Note that Theorem 4 still holds under the weaker memory condition $M = \Omega(n/(D \cdot B)^{2/3})$: instead of maintaining an **IM** boolean array for *all* n nodes it is sufficient to remember subsets of size $\Theta(M)$ and adapt the adjacency lists in **EM** whenever the **IM** data structure is full [8]. This can happen at most $\mathcal{O}((D \cdot B)^{2/3})$ times; each **EM** adjustment of the adjacency lists can be done using $\mathcal{O}((n + m)/(D \cdot B))$ I/Os.

Theorem 4 also holds for graphs that are *nearly Eulerian*, i.e., $\sum_v |\mathit{indegree}(v) - \mathit{outdegree}(v)| = \mathcal{O}(m/n)$: a simple preprocessing can connect nodes with unbalanced degrees via paths of n dummy nodes. The resulting graph G' is Eulerian, has size $\mathcal{O}(n + m)$, and the BFS levels of reachable nodes from the original graph will remain unchanged.

9 Conclusions

We have provided a new BFS algorithm for external memory. For general undirected sparse graphs it is much better than any previous approach. It may facilitate I/O-efficient solutions for other graph problems like demonstrated for some SSSP problem. However, it is unclear whether similar I/O-performance can be achieved on arbitrary directed graphs. Furthermore, it is an interesting open question whether there is a stronger lower-bound for external-memory BFS. Finally, finding an algorithm for depth-first search with comparable I/O-performance would be important.

Acknowledgements. We would like to thank the participants of the GI-Dagstuhl Forschungsseminar “Algorithms for Memory Hierarchies” for a number of fruitful discussions on the “BFS bottleneck”.

References

1. J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.
2. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
3. L. Arge, G. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. In *Proc. 8th Scand. Workshop on Algorithmic Theory*, volume 1851 of *LNCS*, pages 433–447. Springer, 2000.
4. L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. In *Proc. 7th Intern. Workshop on Algorithms and Data Structures (WADS 2001)*, volume 2125 of *LNCS*, pages 471–482. Springer, 2001.
5. L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proc. Workshop on Algorithm Engeneering and Experiments (ALENEX)*, 2000.
6. M. Atallah and U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29(30):330–337, 1984.
7. A. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *Proc. 11th Symp. on Discrete Algorithms*, pages 859–860. ACM-SIAM, 2000.
8. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamasia, D. E. Vengroff, and J. S. Vitter. External memory graph algorithms. In *6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.
9. V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th Symp. on Parallel and Distrib. Processing*, pages 169–177. IEEE, 1996.
10. A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proc. 10th Intern. Symp. on Algorithms and Computations*, volume 1741 of *LNCS*, pages 307–316. Springer, 1999.
11. A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proc. 12th Ann. Symp. on Discrete Algorithms*, pages 89–90. ACM–SIAM, 2001.
12. A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proc. 13th Ann. Symp. on Discrete Algorithms*, pages 372–381. ACM–SIAM, 2002.
13. U. Meyer. External memory BFS on undirected graphs with bounded degree. In *Proc. 12th Ann. Symp. on Discrete Algorithms*, pages 87–88. ACM–SIAM, 2001.
14. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. 10th Symp. on Discrete Algorithms*, pages 687–694. ACM-SIAM, 1999.
15. M. H. Nodine and J. S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, 42(4):919–933, 1995.
16. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two level memories. *Algorithmica*, 12(2–3):110–147, 1994.