Alcom-FT Technical Report Series
ALCOMFT-TR-02-48

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 two versions of a new algorithm which requires only $\mathcal{O}\left((\sqrt{\frac{1}{D \cdot B}} + \sqrt{\frac{n}{m}}) \cdot \frac{n+m}{\sqrt{D \cdot B}} \cdot \log_{M/B} \frac{n+m}{B} \right)$ I/Os (expected or worst-case, respectively). Hence, for $m = \mathcal{O}(n)$, they improve upon the I/O-performance of the best previous algorithm by nearly a factor of $\sqrt{D \cdot B}$. Our approach is fairly simple and we conjecture at least the randomized version to be practical. We also give an improved single-source shortest-paths algorithm for small integer edge weights.

1 Introduction

Breadth-First Search (**BFS**) [15] is a basic graph-traversal method. It decomposes the input graph into levels where level i comprises all nodes that can be reached from a designated source s via a path of i edges. We restrict attention to undirected graphs in this paper. The extension to directed graphs remains open. BFS is used as a subroutine in many graph algorithms; the paradigm of breadth-first search also underlies shortest path computations.

External memory computation is concerned with problems that are too large to fit into main memory. The central issue of external-memory computation is that accessing the secondary memory takes several orders of magnitude longer than performing an internal operation. We use the standard model of external memory computation [18]. 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.

A number of basic computational problems can be solved I/O-efficiently. The most prominent example is external memory sorting [3, 17]: sorting x items of constant size takes $\mathrm{sort}(x) = \Theta(\frac{x}{D \cdot B} \cdot \log_{M/B} \frac{x}{B})$ I/Os.

* 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.

BFS seems to be hard for external memory computation (and also parallel computation). Given sufficiently large main memory, BFS can easily be solved in $\mathcal{O}(n+m)$ time on a graph with n edges and m nodes. The best external memory algorithm known [16] requires $\Theta(n+\text{sort}(n+m))$ I/Os. In this bound the $\Theta(n)$-term results from the unstructured accesses to the adjacency lists.

Recall the standard 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 $\Omega(n)$ I/Os. The algorithm of Munagala and Ranade [16] is much more sophisticated. However, it still pays one I/O for each node.

We first present a new randomized algorithm which requires only

$$\mathcal{O}\left(\sqrt{\frac{n}{m}} \cdot \frac{n+m}{\sqrt{D \cdot B}} \cdot \log_{M/B} \frac{n+m}{B} + \text{sort}(n+m)\right)$$

expected I/Os. Then we modify the preprocessing of this algorithm in order to obtain the same result in the worst case. For sparse graphs with $m = \mathcal{O}(n)$, our approach improves upon the I/O-performance of the best previous algorithm by the factor $\Omega\left(\frac{\sqrt{D \cdot B}}{\log_{M/B}(n+m)}\right)$. Let $m = c \cdot n$ and $L = \log_{M/B}(n+m)$. For $c = m/n \leq D \cdot B/L^2$ the new algorithm is superior to the old algorithm. The number of I/Os for the old algorithm is $\Theta(n) = \Theta(m/c)$, the number of I/Os for our algorithm is $\mathcal{O}(m \cdot L/\sqrt{D \cdot B \cdot c})$ and $m \cdot L/\sqrt{D \cdot B \cdot c} \leq m/c$. Our algorithm is simple and we conjecture it to be practical (at least for the randomized version).

This paper is organized as follows. In Section 2 we review previous work and put our work into context. In Section 3 we outline the new randomized algorithm. 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 describes the required modifications in order to make the approach deterministic. In Section 7 we sketch an extension to a variant of the single-source shortest-paths problem. Finally, Section 8 provides some concluding remarks and open problems.

2 Previous Work

I/O-efficient algorithms for graph-traversal have been considered in, e.g., [1, 2, 4, 5, 7, 8, 10, 12, 13, 11, 14, 16]. In the following we will only discuss results related to BFS; for ease of notation we will also assume $m \geq n-1$.

The currently fastest BFS algorithm for general undirected graphs [16] requires $\Theta(n+\text{sort}(m))$ I/Os. Faster algorithm are only known for special types of graphs: $\mathcal{O}(\text{sort}(n))$-I/O algorithms have been found for BFS on trees [7], grid graphs [6], outer-planar graphs [11], and graphs of bounded tree width [12]. Slightly sublinear I/O was known for undirected graphs with bounded maximum node degree d: the algorithm [14] needs $\mathcal{O}(\frac{n}{\gamma \cdot \log_d(D \cdot B)} + \mathrm{sort}(n \cdot (D \cdot B)^\gamma))$ I/Os and $\mathcal{O}(n \cdot (D \cdot B)^\gamma)$ external space for an arbitrary parameter $0 < \gamma \leq 1/2$. The algorithm can be combined with results from [4] to solve BFS for arbitrary embedded planar graphs using $\mathcal{O}(\frac{n}{\gamma \cdot \log_3(D \cdot B)} + \mathrm{sort}(n \cdot (D \cdot B)^\gamma))$ I/Os. Maheshwari and Zeh [13, 19, 20] proposed I/O-optimal algorithms for a number of fundamental problems on planar graphs; in particular, they show how to compute BFS using $\mathcal{O}(\mathrm{sort}(n))$ I/Os. The algorithms do not need to know an embedding.

The best known lower bound for BFS is $\Omega(\min\{n, \mathrm{sort}(n)\})$ I/Os. It follows from the respective lower bound for the list-ranking problem [8].

We give a new randomized algorithm for BFS on undirected graphs. Our approach needs expected $\mathcal{O}\left(\sqrt{\frac{n}{m}} \cdot \frac{n+m}{\sqrt{D \cdot B}} \cdot \log_{M/B} \frac{n+m}{B} + \mathrm{sort}(n+m)\right)$ I/Os. The new algorithm is particularly suited for sparse graphs: if $m = \mathcal{O}(n)$, it improves upon the I/O-performance of the best previous algorithm by a factor of $\Omega(\sqrt{D \cdot B}/\log_{M/B}(n+m))$. The algorithm is simple and has the chance of being practical (especially the less involved randomized version).

3 High-Level Description of the New BFS Algorithm

Our algorithm refines the algorithm of Munagala and Ranade [16] 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 step partitions the graph into disjoint connected subgraphs $\mathcal{S}_i$, $0 \leq i \leq K$ 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 partition is constructed by choosing seed nodes independently and uniformly at random with probability μ (for technical reasons, s is also made a seed) and running a BFS starting from all seed nodes. Then the expected number of seed nodes is $K = \mathcal{O}(1 + \mu \cdot n)$ and the expected shortest-path distance between any two nodes of a subgraph is at most $2/\mu$. The expected time for constructing the partition is $\mathcal{O}(\mathrm{sort}(n+m)/\mu)$.

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 $\mathcal{H}$ 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 $\mathcal{O}\left((K + \frac{n+m}{D \cdot B}) \cdot \log_{M/B} \frac{n+m}{B}\right)$ 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 $\mathcal{O}\left((K + \frac{n+m}{D \cdot B}) \cdot \log_{M/B} \frac{n+m}{B} + \mathrm{sort}(n+m)/\mu\right) = \mathcal{O}\Big((1 + \frac{(1+\mu \cdot n) \cdot D \cdot B}{n+m} + \frac{1}{\mu}) \cdot \mathrm{sort}(n+m)\Big)$ on the expected number of I/Os for the second phase. Choosing $\mu = \min\left\{1, \sqrt{\frac{n+m}{n \cdot D \cdot B}}\right\}$ gives our bound.

4 The Algorithm of Munagala and Ranade

We review the BFS algorithm of Munagala and Ranade [16], 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 $\mathcal{O}(\text{sort}(n+m))$ I/Os: BFS Numbers $\rightarrow$ BFS Tree $\rightarrow$ BFS Levels $\rightarrow$ 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 $\mathcal{O}\left(|L(t-1)| + |A(t)|/(D \cdot B)\right)$ 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 $\mathcal{O}(\text{sort}(|A(t)|)$ I/Os. The resulting set $A'(t)$ is still sorted.

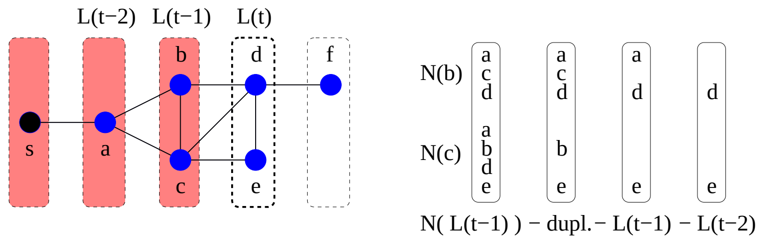

Fig. 1. A phase in the BFS algorithms of Munagala and Ranade [16]. Level $L(t)$ is composed out of the disjoint neighbor vertices of level $L(t-1)$ excluding those vertices already existing in either $L(t-2)$ or $L(t-1)$.

Now MR_BFS computes $L(t) := A'(t) \setminus \{L(t-1) \cup L(t-2)\}$. Figure 1 provides an example. Filtering out the nodes already contained in the sorted lists $L(t-1)$ or $L(t)$ is possible by parallel scanning. Therefore, this step can be done using $\mathcal{O}(\text{sort}(|A(t)| + |L(t-1)| + |L(t-2)|))$ I/Os. Since $\sum_t |A(t)| = \mathcal{O}(m)$ and $\sum_t |L(t)| = \mathcal{O}(n)$, MR_BFS requires $\mathcal{O}(n + \text{sort}(n+m))$ 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 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 1 ([16]). *BFS on arbitrary undirected graphs can be solved using* $\mathcal{O}(n + \mathrm{sort}(n+m))$ *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 run the randomized $\mathcal{O}(\mathrm{sort}(n+m))$-I/O connected-components algorithm of [1] and only keep the nodes and edges of the component C_s that contains the root 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 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 sub-graphs $\mathcal{S}_i$ 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 μ where

$$\mu = \min\left\{1, \sqrt{\frac{n+m}{n \cdot D \cdot B}}\right\}.$$

This choice of μ minimizes the total cost of the algorithm as we will see later. Additionally, we make sure that the root 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 a 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. Each phase can be implemented by a constant number of scan- and sorting-steps on an adjacency-lists representation of the yet unexplored graph:

At the beginning of a phase, the adjacency lists of the nodes lying on the boundaries of the current partitions are marked to be active; they carry the label of their master node. A scan extracts 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). 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 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 $X := \mathcal{O}(\sum_{v \in V} 1/\mu \cdot (1 + \text{degree}(v))) = \mathcal{O}((n+m)/\mu)$. Thus the preprocessing requires $\mathcal{O}(\text{sort}(X)) = \mathcal{O}(\text{sort}(n+m)/\mu)$ 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 preprocessing of* FAST_BFS *requires $\mathcal{O}(\text{sort}(n+m)/\mu)$ 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 $\mathcal{H}$. It comprises the unused parts of the $\mathcal{F}_i$ which contain a node in the current level $L(t-1)$. We initialize $\mathcal{H}$ with $\mathcal{F}_0$. Thus, in particular, $\mathcal{H}$ contains the adjacency list of the root node s of level $L(0)$. The nodes of each created BFS levels 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 $A(t)$. It then removes duplicates and constructs $L(t)$ as $A(t) \setminus \{L(t-2) \cup L(t-1)\}$. Having constructed $L(t)$ it knows which $\mathcal{F}_i$ to add to $\mathcal{H}$.

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*

$$\mathcal{O}\left((1 + \frac{\mu \cdot n \cdot D \cdot B}{n+m} + \frac{1}{\mu}) \cdot \mathrm{sort}(n+m)\right)$$

expected I/Os.

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 $\mathcal{H}$, $L(t-2)$, and $L(t-1)$ will be scanned and sorted 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, sorting and 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 sort and scan x data items over the whole execution of FAST_BFS is bounded by $\mathcal{O}(\mathrm{sort}(x))$. Similar to MR_BFS, sorting and scanning all BFS levels takes $\mathcal{O}(\sum_i \mathrm{sort}(|L(i)|)) = \mathcal{O}(\mathrm{sort}(n))$ I/Os.

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, several subfiles $\mathcal{F}_i$ may be merged with $\mathcal{H}$ in one big merge operation; which takes at most $\mathcal{O}(\log_{M/B} \frac{n+m}{B})$ stages). However, 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

$$\mathcal{O}\left(\sum_i (1 + \frac{|\mathcal{F}_i|}{D \cdot B}) \cdot \log_{M/B} \frac{n+m}{B}\right) = \mathcal{O}\left(K \cdot \log_{M/B} \frac{n+m}{B} + \mathrm{sort}(n+m)\right)$$
$$= \mathcal{O}\left((1 + \frac{K \cdot D \cdot B}{n+m}) \cdot \mathrm{sort}(n+m)\right)$$

Also $\mathbf{E}[K] = 1 + \mu \cdot n$ and $\frac{(1+\mu \cdot n) \cdot D \cdot B}{n+m} \leq 1 + \frac{\mu \cdot n \cdot D \cdot B}{n+m}$.

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 of $2/\mu$ rounds. This follows from the fact that the expected shortest-path distance between any two nodes of a subgraph is $2/\mu$: 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 $\mathcal{O}(\text{sort}(n+m)/\mu)$ expected I/Os. By the same argumentation, each adjacency list in $\mathcal{H}$ takes part in at most $\mathcal{O}(1/\mu)$ sort- and scan-steps for the generation of $N(L(\cdot))$ and $L(\cdot)$. □

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

Theorem 2. *External memory BFS on undirected graphs can be done using* $\mathcal{O}\left(\sqrt{\frac{n}{m}} \cdot \frac{n+m}{\sqrt{D \cdot B}} \cdot \log_{M/B} \frac{n+m}{B} + \text{sort}(n+m)\right)$ *expected I/Os.*

Proof: By our lemmas the expected number of I/0s is bounded by

$$\mathcal{O}\left((1 + \frac{\mu \cdot n \cdot D \cdot B}{n+m} + \frac{1}{\mu}) \cdot \text{sort}(n+m)\right).$$

The expression in the first bracket is approximately 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 [16] using $\mathcal{O}((1+\log\log(D \cdot B \cdot n/m)) \cdot \text{sort}(n+m))$ I/Os. The same amount of I/O suffices to compute a (minimum) spanning tree T_s for C_s [4].

After T_s has been built, the preprocessing constructs an Euler Tour [9] around T_s using a constant number of sort- and scan-steps [8]. 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 [8]. This causes $\mathcal{O}(\text{sort}(n))$ 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 multiple node appearances and make sure that each node of C_s is part of exactly one subgraph $\mathcal{S}_i$, namely 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 reordered adjacency-list files $\mathcal{F}_i$; this is done as in the old preprocessing and takes another $\mathcal{O}(\text{sort}(n+m))$ 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 $L(t)+1/\mu-1$. Therefore, the adjacency list of v in $\mathcal{F}_i$ will be kept in $\mathcal{H}$ for at most $1/\mu$ BFS levels. The bound on the total number of I/Os follows from the fact that $\mathcal{O}((1+\log\log(D \cdot B \cdot n/m)) \cdot \mathrm{sort}(n+m)) = \mathcal{O}(\sqrt{\frac{n}{m}} \cdot \frac{n+m}{\sqrt{D \cdot B}} \cdot \log_{M/B} \frac{n+m}{B} + \mathrm{sort}(n+m))$.

Theorem 3. *External memory BFS on undirected graphs can be done using* $\mathcal{O}\left(\sqrt{\frac{n}{m}} \cdot \frac{n+m}{\sqrt{D \cdot B}} \cdot \log_{M/B} \frac{n+m}{B} + \mathrm{sort}(n+m)\right)$ *I/Os in the worst case.*

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. A simple extension of Fast_BFS needs

$$\mathcal{O}\left(\frac{n+m}{\sqrt{D \cdot B}} \cdot \left(\frac{W}{\sqrt{D \cdot B}} + \sqrt{\frac{W \cdot n}{m}}\right) \cdot \log_{M/B} \frac{n+m}{B}\right)$$

expected I/Os. Thus, for sparse graphs and constant W the resulting algorithm Fast_SSSP requires $\mathcal{O}(\frac{n}{\sqrt{D \cdot B}} \cdot \log_{M/B} \frac{n}{B})$ expected I/Os.

For integer weights in $\{1,\ldots,W\}$, the maximum shortest-path distance of an arbitrary reachable node to the root 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 $L(1)$, $L(2)$, $\ldots$; 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 $\mathcal{O}\left((W/\mu) \cdot \mathrm{sort}(n+m)\right)$. The choice $\mu = \min\left\{1, \sqrt{\frac{W \cdot (n+m)}{n \cdot D \cdot B}}\right\}$ balances the costs of the various phases and the stated bound results.

8 Conclusions

We have provided a new BFS algorithm for external memory. For general 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, our results are restricted to undirected graphs. 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.

References

1. J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.
2. P. K. Agarwal, L. Arge, T. M. Murali, K.R. Varadarajan, and J.S. Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking. In *Proc. 9th Symp. on Discrete Algorithms*, pages 117–126. ACM-SIAM, 1998.
3. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
4. 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.
5. 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.
6. 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.
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. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. E. F. Moore. The shortest path through a maze. In *Proc. Intern. Symp. on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

16. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. 10th Symp. on Discrete Algorithms*, pages 687–694. ACM-SIAM, 1999.
17. 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.
18. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two level memories. *Algorithmica*, 12(2–3):110–147, 1994.
19. N. Zeh. I/O-efficient planar separators and applications. Technical Report TR-01-02, School of Computer Science, Carleton University, Ottawa, Canada, 2001.
20. N. Zeh. I/O-optimal planar embedding using graph separators. Technical Report TR-01-07, School of Computer Science, Carleton University, Ottawa, Canada, 2001.