Algorithmica (2002) 32: 437–458
DOI: 10.1007/s00453-001-0088-5

Algorithmica
© 2002 Springer-Verlag New York Inc.

A Functional Approach to External Graph Algorithms[1]

J. Abello,[2] A. L. Buchsbaum,[2] and J. R. Westbrook[3]

Abstract. We present a new approach for designing external graph algorithms and use it to design simple, deterministic and randomized external algorithms for computing connected components, minimum spanning forests, bottleneck minimum spanning forests, maximal independent sets (randomized only), and maximal matchings in undirected graphs. Our I/O bounds compete with those of previous approaches. We also introduce a semi-external model, in which the vertex set but not the edge set of a graph fits in main memory. In this model we give an improved connected components algorithm, using new results for external grouping and sorting with duplicates. Unlike previous approaches, ours is purely functional—without side effects—and is thus amenable to standard checkpointing and programming language optimization techniques. This is an important practical consideration for applications that may take hours to run.

Key Words. Connected components, External memory algorithms, Graph algorithms, Maximal matchings, Maximal independent sets, Minimum spanning trees, Functional programming.

1. Introduction. Because classical algorithms often do not scale when data exceed main memory limits, recent attention has focused on algorithms that process data in external storage (disk and tape) [1], [6], [36]. While some external algorithms (e.g., for sorting [2]) closely resemble their RAM analogues, others seem to require different approaches. Graph algorithms for RAMs, in particular, seem poorly suited for direct extension to external memory, because of the lack of locality in graph data. Current approaches for designing external graph algorithms use either PRAM simulation techniques [13], or else external data structures [4], [26] that do not completely address the I/O implications of graph traversal.

We present a new divide-and-conquer approach for designing external graph algorithms, which is simple to describe and implement. No sophisticated data structures are needed. When unwinding the divide-and-conquer recursions, our algorithms effect successions of graph transformations that reduce to sorting, selection, and a recursive bucketing technique. We apply our techniques to devise external algorithms for computing connected components, minimum spanning forests (MSFs), bottleneck minimum spanning forests (BMSFs), maximal independent sets, and maximal matchings in undirected graphs.

We focus on producing algorithms that are purely functional. That is, each algorithm is specified as a sequence of functions applied to input data and producing output data,

[1] An extended abstract of this work appears in *Proc.* 6*th European Symposium on Algorithms*, Venice, 1998.
[2] AT&T Labs, Shannon Laboratory, 180 Park Ave., Florham Park, NJ 07932, USA. {abello, alb}@research.att.com.
[3] 20th Century Television, Los Angeles, CA 90025, USA. jwestbrook@acm.org. Work completed while a member of AT&T Labs.

Received February 12, 1999; revised April 10, 1999. Communicated by G. N. Frederickson.
Online publication November 23, 2001.

with the property that information, once written, remains unchanged. The function is then said to have no "side effects." A functional approach has several benefits. External memory algorithms may run for hours or days in practice. As we argue below, the lack of side effects on the external data allows standard checkpointing techniques to be applied [27], [32], increasing the reliability of any real application. A functional approach is also amenable to general purpose programming language transformations that can reduce running time. (See, e.g., [37].)

Formally, we adapt the I/O model of complexity as defined by Aggarwal and Vitter [2]. We parameterize a problem instance as follows:

$$\begin{aligned} N &= \text{number of items in the instance,} \\ M &= \text{number of items that can fit in main memory,} \\ B &= \text{number of items per disk block.} \end{aligned}$$

A typical compute server might have $M \approx 10^9$ and $B \approx 10^3$. In general, $1 < B < M/2$, and $M < N$. For some of our randomized algorithms, we also assume that $B = O(N/\log^{(i)} N)$ for some fixed integer $i > 0$, where $\log^{(0)} N = N$, and $\log^{(i)} N = \log\log^{(i-1)} N$ for $i > 0$. We present our algorithms in the single-disk model; extending them to parallel disks remains open.

For a graph, we define V to be the number of vertices, E to be the number of edges, and $N = V + E$. (We abuse notation and also use V and E to be the actual vertex and edge sets; the context will clarify any ambiguity.)

Because our external algorithms rely on scanning and sorting as primitives, when possible results are expressed in terms of $\mathit{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$, the number of I/Os needed to sort N items [2], and $\mathit{scan}(N) = \lceil N/B \rceil$, the number of I/Os needed to transfer N contiguous items between disk and internal memory. The I/O model stresses the importance of disk accesses over computation for large problem instances. In particular, time spent computing in main memory is not counted. A goal of external algorithm design is to replace factors of N in the time complexity by factors of N/B in the I/O complexity (utilizing an entire disk block once it is read) and $\log_2$ terms in the time complexity by $\log_{M/B}$ terms in the I/O complexity (dividing problems into M/B subproblems at a time).

The *functional* I/O (*FIO*) *model* is parameterized as above, but operations are restricted to make only functional transformations to data, which do not change the input. Once a disk block is allocated and written, its contents cannot be changed. This imposes a sequential write-once discipline on disk writes. When results of intermediate computations are no longer needed, space is reclaimed, e.g., through garbage collection. The maximum disk space active at any one time is used to measure the space complexity. All of our algorithms use only linear space (i.e., $O(N/B)$ disk blocks).

Consider the implication of the functional approach on checkpointing. The only information recorded about external files by standard checkpointing techniques [27], [32] is the file-descriptor table in the kernel: i.e., which files are assigned to which file descriptors, and the current seek positions within each file. Overwrites to disk blocks after the most recent checkpoint therefore interfere with the correctness of the recovery process, because data is not preserved as at the time of the checkpoint. Alternative checkpointing methods copy open files during checkpoints, but this can be prohibitively expensive if the files are massive (as in the case of external memory algorithms), even with lazy copying techniques [38]. A functional external algorithm, therefore, lends itself to correct checkpointing, increasing its robustness.

1.1. *Problem Definitions*. In all cases we assume that the input graph, G, is presented as an unordered list of edges, each edge a pair of vertices plus possibly a weight. We consider the following problems:

Connected components. A *connected component* is the edge set induced by a maximal set of vertices such that each pair of vertices is connected by a path in G. The output is a *delineated list* of edges, $\{C_1, C_2, \ldots, C_k\}$, where k is the number of components. Each C_i is the list of edges in component i, and the output is the file of components catenated together, with a separator record between adjacent components.

Minimum spanning forests. A *minimum spanning forest* (*MSF*) is a spanning forest that minimizes the sum of the weights of the edges. The output is a list of edges in the forest, delineated by trees.

Bottleneck minimum spanning forests. A *bottleneck minimum spanning forest* (*BMSF*) is a spanning forest that minimizes the weight of the maximum edge. The output is a list of edges in the forest, delineated by trees.

Maximal matching. A *maximal matching* is a maximal set of edges such that no two edges share a common vertex. The output is a list of edges in the matching.

Maximal independent set. A *maximal independent set* is a maximal set of vertices such that no two vertices are adjacent. The output is a list of vertices in the independent set.

1.2. *Results*. Our main contribution is the functional approach to external memory algorithm design, which provides a uniform framework in which we can derive simple, robust algorithms that match or slightly improve upon previous results for a host of graph problems. Table 1 summarizes our results. For the external connected components, MSF, and BMSF problems, the only known lower bound is $\Omega(sort(V))$ [13]. (The BMSF lower bound derives from that on connected components, given the stipulation of the output.) No non-trivial lower bounds are known for external maximal matchings or maximal independent sets.

Table 1. I/O bounds for our functional external algorithms.

	Deterministic	Randomized	
Problem	I/O bound	I/O Bound	With probability
Connected components	$O(sort(E) + \frac{E}{V} sort(V) \log_2 \frac{V}{M})$	$O(sort(E))$	$1 - e^{\Omega(E)}$
MSFs	$O(sort(E) + \frac{E}{V} sort(V) \log_2 \frac{V}{M})$	$O(sort(E))$	$1 - e^{\Omega(E)}$
BMSFs	$O(sort(E) + \frac{E}{V} sort(V) \log_2 \frac{V}{M})$	$O(sort(E))$	$1 - e^{\Omega(E)}$
Maximal matchings	$O(\frac{E}{V} sort(V) \log_2 \frac{V}{M})$	$O(sort(E))$	$1 - \varepsilon$ for any fixed ε
Maximal independent sets		$O(sort(E))$	$1 - \varepsilon$ for any fixed ε

Chiang et al. [13] give deterministic connected components and MSF algorithms that use $O(sort(E)\log_2(V/M))$ I/Os. The asymptotic I/O complexities of our deterministic connected components and MSF algorithms are slightly better, because $(E/V)sort(V) = (E/B)\log_{M/B}(V/B)$. Kumar and Schwabe [26] give algorithms for breadth-first search (BFS, which can compute connected components) and MSFs that perform $O(V+sort(E)\log_2(M/B))$ and $O(sort(E)\log_2 B+scan(E)\log_2 V)$ I/Os, respectively. Our connected components algorithm is asymptotically better when $V < M^2/B$, and our MSF algorithm is asymptotically better when $V < MB$. While the above algorithms of Chiang et al. [13] are functional, those of Kumar and Schwabe [26] are not. Our randomized connected components and MSF algorithms match the I/O performance given by Chiang et al. [13].

The BMSF, maximal independent set, and maximal matching results are new. The BMSF result derives directly from the MSF result; we state it to illuminate an open problem that leads to a distinction in the relationship between the two problems in the internal and external memory settings. The maximal independent set and matching results come from straightforward externalizations of internal algorithms, but they emphasize the utility of the functional design approach.

A recent paper by Munagala and Ranade [31] gives a deterministic $O(\max\{1, \log\log(VBP/E)\}(E/V)sort(V))$ I/O algorithm for connected components, where P is the number of parallel disks. While this improves upon our deterministic I/O bound for connected components, some of our techniques (in particular the relabeling techniques we describe in Section 3) can be used to simplify their implementation.

Finally, we consider a *semi-external* model for graph problems, in which the vertices but not the edges fit in memory. This is not uncommon in practice, and when vertices can be kept in memory, significantly more efficient algorithms are possible. We design algorithms, based on distribution sort [24], for external grouping and sorting with duplicates and apply them to produce better I/O bounds for the semi-external case of connected components.

There is much other work on external graph problems, in particular on depth- and breadth-first search [9], [13], [26], [31], topological sorting [13], single-source shortest paths [26], and transitive closure [13], [35]. Many of these rely heavily on graph traversal (following an edge from one vertex to the next) and/or use heap data structures (discussed in Section 2) and are not immediately amenable to our functional approach.

In Section 2 we sketch two previous approaches for designing external graph algorithms. In Section 3 we describe our functional approach and detail a suite of simple graph transformations. In Section 4 we apply our approach to design new, simple, deterministic algorithms for computing connected components, MSFs, BMSFs, and maximal matchings. In Section 5 we give randomized variants of our algorithms, including one for maximal independent sets, with improved I/O bounds. In Section 6 we consider semi-external graph problems and give improved I/O bounds for the semi-external case of connected components. We conclude in Section 7.

2. Previous Approaches.

While sorting and other "data rearrangement" problems are well suited to external memory [2], graph problems do not generally present the data locality needed to realize straightforward and efficient external-memory extensions of classical algorithms. The on-line nature of graph traversal makes it difficult both to utilize a full disk block of edges once read (and so to reduce N time bounds to N/B I/O bounds) and to divide a problem into many independent subproblems (and so to reduce $\log_2 x$ time bounds to $\log_{M/B} x$ I/O bounds). External algorithms for some graph problems have been designed, however, and here we sketch two current approaches for doing so. (Munagala and Ranade [31] use neither approach to design their connected components algorithm, exploiting an efficient undirected BFS algorithm instead. They do use some results from Chiang et al. [13], in particular pointer jumping, which we discuss in Section 3.)

2.1. *PRAM Simulation*. Chiang et al. [13] show how to simulate a CRCW PRAM algorithm using one processor and an external disk, thus giving a general method for constructing external graph algorithms from PRAM graph algorithms.

Given a CRCW PRAM algorithm on N processors, the simulation maintains on disk (1) a copy of main memory in an array, A, sorted by memory address, and (2) a state array, T, of N elements. Let $A[i]$ (resp., $T[i]$) be the ith element of A (resp., T). Location $T[i]$ contains the ($O(1)$ size) current state of processor i.

A step of the PRAM algorithm is simulated as follows. Each step is begun with a list D of tuples $(d(i), i)$ where $d(i)$ is the memory address that processor i will read in this step. D is then sorted by the first component (memory address) of each record. We can then scan A and D in tandem, writing a list R of records $(r(i), i)$, where $r(i) = A[d(i)]$. List R is then sorted by second component (processor number), and the sorted R and T are scanned in tandem. For each processor i, $T[i]$ is updated to $T'[i]$ using the read value $r(i)$, and a list W of records $(d(i), w(i))$ is written, where $d(i)$ is the memory address written by processor i in the simulated step and $w(i)$ is the value written. At the same time, the list D of read locations for the next step is generated. List W is sorted by first component and scanned in tandem with A to produce A', the simulated contents of main memory after the PRAM step; T' contains the updated processor states.

Each step of the PRAM algorithm thus requires three scans and three sorts of arrays of size $|T|$, and two scans of A. Typically, therefore, a PRAM algorithm using N processors and N space to solve a problem of size N in time t can be simulated in external memory by one processor using $O(t \cdot sort(N))$ I/Os. An extension to the simulation produces the same $O(t \cdot sort(N))$ I/O bound for any N-processor, $O(N)$-space PRAM algorithm such that, after each of $\log N$ stages, each of time t, the number of active processors and memory cells that will ever be used again is reduced by a constant factor.

The PRAM simulation works in the FIO model, if each step writes new copies of T and A. To the best of our knowledge, however, no algorithm based on the simulation has been implemented. A direct implementation would require not only a practical PRAM algorithm but also either a meticulous direct implementation of the corresponding external memory simulation or a suitable low-level machine description of the PRAM algorithm together with a general simulation tool. Rather, PRAM simulation is typically used to prove the existence of an external memory algorithm of a given I/O complexity. For example, Chiang et al. [13] describe the application of PRAM simulation to derive $O(sort(E) \log_2(V/M))$-I/O connected components and MSF algorithms from the work of Chin et al. [14]. The functional approach captures many of the results possible with PRAM simulation and also produces robust algorithms, amenable to checkpointing.

2.2. *Buffering Data Structures*. Another recent approach is based on external variants of classical internal data structures. Arge [4] introduces *buffer trees*, which support sequences of insert, delete, and deletemin operations on N elements in $O((1/B)\log_{M/B}(N/B))$ amortized I/Os each. Kumar and Schwabe [26] introduce a variant of the buffer tree, achieving the same heap bounds as Arge. Later works by Brodal and Katajainen [8] and Fadel et al. [17] focus on improving the I/O complexity for individual operations. These bounds are optimal, since the heaps can be used to sort externally. Kumar and Schwabe [26] also introduce external *tournament trees*. The tournament tree maintains the elements 1 to N, each with a key, subject to the operations delete, deletemin, and update. Deletemin returns the element of minimum key. Update takes a pair (x, k), and sets the key of x to $\min\{\text{key}(x), k\}$: i.e., it reduces the key of x to k if and only if the current key exceeds k. Each tournament tree operation takes $O((1/B)\log_2(N/B))$ amortized I/Os.

The data structures of Arge [4] and Kumar and Schwabe [26] closely resemble classical, tree-based internal heaps. Each node maintains a large buffer (e.g., of size cM for some $0 < c < 1$) that stores operations. An operation (insert, delete, etc.) is performed by adding it to the buffer of the root node. When a buffer becomes full, its operations are percolated to the appropriate children buffers. An operation is effected when it "meets" its operand in the tree. The maintenance procedures on the data structures are intuitively simple but can involve many implementation details. In general, internal memory limitations require that buffer emptying be performed in two stages (emptying half of a buffer, recursively considering buffers of children, and then repeating to empty the rest of the buffer), although in practice, simple one-stage versions can be implemented for a restricted set of operations. Fadel et al. [17] describe a different tree-based data structure. Rather than maintain a tree, Brodal and Katajainen [8] maintain a set of sorted lists, similarly subject to buffered updates, which engender incremental merges of the lists.

None of these data structures is functional. The tree-based structures [4], [17], [26] require in situ replacement of nodes stored on disk; Brodal and Katajainen's [8] update scheme also relies on in situ replacement of disk blocks, to effect efficient catenation of lists undergoing merging. The node-copying techniques of Driscoll et al. [15] could be used to make them functional: any time a disk block is to be overwritten, a new, suitably modified, copy is created instead. This incurs significant extra I/O overhead, however, because any blocks pointing to the modified block must be likewise updated to point to the new copy; such updates thus percolate to the root of the structure. The number of I/Os required to modify a node in a tree (resp., list), therefore, would be proportional to the depth of the tree (resp., length of the list).

Finally, while such data structures excel in computational geometry applications, they are hard to apply to external graph algorithms. Consider computing an MSF. The classical greedy algorithm maintains a growing MSF, F, and a heap of vertices outside F. Each step performs a deletemin to attach the closest vertex, v, to F. Each neighbor, w, of v is considered in turn. If w becomes closer to F by way of v, then w's key in the heap is decreased. (If the heap does not admit an appropriate update operation, w is deleted and re-inserted into the heap.)

There are two obstacles to this approach in the external version of the algorithm. First, finding the neighbors of v is non-trivial, involving the creation of and indexing into an adjacency list, which incurs at least one I/O for each vertex. Second, given a neighbor w, determining the current key of w (distance of w to F) is problematic, for the data structures do not permit arbitrary searches. Without the key, however, it is impossible to know whether or not to perform the update operation in the first place.

Intuitively, while the update operations on the external data structures can be buffered, yielding efficient amortized I/O complexity, standard applications of these data structures in graph algorithms require certain queries (e.g., key finding) to be performed on-line. Current applications of these data structures thus require ancillary data structures to obviate this problem, increasing the I/O and implementation complexity. For example, Kumar and Schwabe [26] extend the classical undirected BFS and greedy MSF algorithms to the external setting. They attack the key-finding problem using both the tournament tree, with its particular update operation, and a heap, with keys encoded to record the update strategy that should be effected as the algorithm progresses.

3. Functional Graph Transformations. In this paper we utilize a divide-and-conquer paradigm based on a few graph transformations that, for many problems, preserve certain critical properties. We implement each stage in our approach using simple and efficient techniques: sorting, selection, and bucketing. We illustrate our approach with connected components. Let $G = (V, E)$ be a graph. Let $CC(G) \subseteq V \times V$ be a forest of rooted stars (trees of height one) representing the connected components of G. That is, if $r_G(v)$ is the root of the star containing v in $CC(G)$, then $r_G(v) = r_G(u)$ if and only if v and u are in the same connected component in G.

We define a contraction operation on vertex pairs: given a graph G, *contracting* a pair of vertices, $\{x, y\}$, adds a new vertex, z, called a *supervertex*, to G, deletes vertices x and y, and replaces all edges of the form $\{u, x\}$ and $\{u, y\}$ with the corresponding edges $\{u, z\}$, discarding any loop edges $\{z, z\}$. (This generalizes the usual notion of contraction, by allowing the contraction of vertices that are not adjacent in G.) Consider a set, E', of vertex pairs, and let G/E' denote the result of contracting all vertex pairs in E'. Intuitively, the contractions happen simultaneously; e.g., if $\{x, y\}$ and $\{y, z\}$ exist in E', then x, y, and z are contracted into the same supervertex. More precisely, let $E' = \{\{x_1, y_1\}, \ldots, \{x_\ell, y_\ell\}\}$. Let $G' = G/\{\{x_1, y_1\}\}$, and let z be the supervertex into which x_1 and y_1 were contracted. Consider the remaining set $E'' = E' \setminus \{\{x_1, y_1\}\}$, such that each occurrence of x_1 and y_1 in pairs in E'' is replaced by z. Then G/E' is defined to be G'/E'', and $G/\emptyset$ is defined to be G.

For any $x \in V$, let $s(x)$ denote the supervertex in $G' = G/E'$ into which x is ultimately contracted; let $s(x) = x$ if x is not in any pair in E'. The following lemma shows how contraction preserves connected components.

LEMMA 3.1. *If each pair in E' contains two vertices in the same connected component in G, then, for any $u, v \in V$, u and v are in the same connected component in G if and only if $s(u)$ and $s(v)$ are in the same connected component in G'.*

PROOF. Consider any edge $\{u, v\}$ in G. By definition of contraction, either $s(u) = s(v)$ or else $\{s(u), s(v)\}$ is an edge in G'. Thus, for any path connecting two vertices, a and b, in G, there is a corresponding (perhaps null) path connecting $s(a)$ and $s(b)$ in G'.

Now consider any edge $\{\mu, \nu\}$ in G'. Let $s^{-1}(\mu)$ (resp., $s^{-1}(\nu)$) be the set of vertices in G that were contracted into μ (resp., ν). By assumption, all vertices in each of $s^{-1}(\mu)$

and $s^{-1}(\nu)$ are in the same connected component. By definition of contraction, therefore, there exists a path in G connecting each vertex in $s^{-1}(\mu)$ with each vertex in $s^{-1}(\nu)$. Thus, for any path connecting two vertices, α and β, in G', there is a corresponding path connecting a and b in G for all $a \in s^{-1}(\alpha)$ and all $b \in s^{-1}(\beta)$. □

We also define a relabeling operation. Given a rooted forest F as an unordered sequence of oriented tree edges $\{(p(v), v), \ldots\}$, and an edge set I, *relabeling* produces a new edge set $RL(F, I) = \{\{r(u), r(v)\} : \{u, v\} \in I\}$, where $r(x) = p(x)$ if $(p(x), x) \in F$, and $r(x) = x$ otherwise. That is, for each edge $\{u, v\} \in I$, each of u and v is replaced by its respective parent, if it exists, in F.

Using contraction and relabeling, we derive the following simple algorithm to compute $CC(G)$, represented as an edge list:

```
Algorithm CC
1. Let $E_1$ be any half of the edges of $G$; let $G_1 = (V, E_1)$.
2. Compute $CC(G_1)$ recursively.
3. Let $G' = G/CC(G_1)$.
4. Compute $CC(G')$ recursively.
5. $CC(G) = CC(G') \cup RL(CC(G'), CC(G_1))$.
```

LEMMA 3.2. *Algorithm* **CC** *correctly computes* $CC(G)$, *a forest of rooted stars corresponding to the connected components of* G.

PROOF. Since $CC(G_1)$ is a forest of stars, we can assume without loss of generality that, when contracting by $CC(G_1)$, we use the root of each star as the canonical supervertex for its component. Each vertex in G' thus corresponds directly to its counterpart in G, which is either the root of a star in $CC(G_1)$ or a vertex not included in any edge in G_1.

By definition, two vertices are in the same star in $CC(G_1)$ only if they are in the same connected component in G. Lemma 3.1 thus shows that two vertices are in the same connected component in G' if and only if they are in the same connected component in G. Consider $F = CC(G') \cup CC(G_1)$. F is a forest of depth two, which, by the above arguments, represents the connected components of G. The relabeling $RL(CC(G'), CC(G_1))$ effects a round of pointer jumping on F: each vertex of depth two becomes a child of the root of its tree. Thus $CC(G)$ as defined in step 5 is a forest of rooted stars. □

We generalize the above into a purely functional approach to design external graph algorithms. Formally, let $f_{\mathcal{P}}(G)$ denote the solution to a graph problem P on an input graph $G = (V, E)$. For a subgraph $G_1 = S(G) \subseteq E$ of G, let T_1 be a transformation that combines G and the solution $f_{\mathcal{P}}(G_1)$ to create a new subgraph, G_2. Let T_2 be a transformation that maps the solutions $f_{\mathcal{P}}(G_1)$ and $f_{\mathcal{P}}(G_2)$, to a solution to G. We summarize the approach as follows:

```
1. $G_1 \leftarrow S(G)$;
2. $G_2 \leftarrow T_1(G, f_{\mathcal{P}}(G_1))$;
3. $f_{\mathcal{P}}(G) = T_2(G, G_1, G_2, f_{\mathcal{P}}(G_1), f_{\mathcal{P}}(G_2))$.
```

In the connected components example, $f_{\mathcal{P}}(G)$ was the forest of rooted stars corresponding to the connected components of G; $G_1 = S(G)$ was a subset of half the edges of G; $G_2 = T_1(G, f_{\mathcal{P}}(G_1))$ was the contraction of G with respect to the connected components of $S(G)$; and $T_2(G, G_1, G_2, f_{\mathcal{P}}(G_1), f_{\mathcal{P}}(G_2))$ was the re-expansion of the rooted stars in $f_{\mathcal{P}}(G_2)$ into the solution $f_{\mathcal{P}}(G)$.

It is critical to realize that the transformations must preserve both solutions and non-solutions to P. For example, contraction preserves the connected components of a graph, so we can use it as above in a divide-and-conquer scheme to compute connected components. On the other hand, while contraction preserves planarity, it does not preserve non-planarity: non-planar graphs contain planar minors. Thus, it is not clear that we could devise an external memory planarity testing algorithm using contraction in such a divide-and-conquer scheme. In general, a lemma analogous to Lemma 3.1 must be demonstrated with respect to the transformations used.

Our approach is functional if S, T_1, and T_2 can be implemented without side effects on their arguments. Below we show how selection, relabeling, contraction, and (vertex and edge) deletion can be implemented functionally. In the following sections we use these tools to design functional external algorithms for computing connected components, MSFs, BMSFs, maximal independent sets, and maximal matchings.

3.1. *Selection*. Let I be a list of items with totally ordered keys. `Select`(I,k) returns the kth biggest element from I, including multiplicity; i.e., $|\{x \in I : x < \texttt{Select}(I, k)\}| < k$ and $|\{x \in I : x \leq \texttt{Select}(I, k)\}| \geq k$. We adapt the classical algorithm for `Select`(I,k) [3]. Aggarwal and Vitter [2] use the same approach to select partitioning elements for distribution sort:

1. Partition I into cM-element subsets, for some $0 < c < 1$.
2. Determine the median of each subset in main memory. Let S be the set of medians of the subsets.
3. $m \leftarrow \texttt{Select}(S, \lceil S/2 \rceil)$.
4. Let I_1, I_2, I_3 be the sets of elements less than, equal to, and greater than m, respectively.
5. If $|I_1| \geq k$, then return $\texttt{Select}(I_1, k)$.
6. Else if $|I_1| + |I_2| \geq k$, then return m.
7. Else return $\texttt{Select}(I_3, k - |I_1| - |I_2|)$.

LEMMA 3.3. `Select`*(I,k) can be performed in* $O(scan(|I|))$ *I/Os in the FIO model.*

PROOF. Let $T(|I|)$ be the number of I/Os needed to perform `Select`(I,k) for any k. The analysis mimics that of the time taken by the classical algorithm [3].

The partitioning of I into cM-element subsets and calculation of the medians can be performed in one scan of I. Similarly, given m, the formation of the sublists I_1, I_2, and I_3 can be performed in one scan of I. Computing m recurses on I/cM elements. One-half of the elements in S are no less than m; each such element, m', was the median of a subset, and so one-half of the elements in that subset were no less than m'. Thus, at least one-quarter of the elements in I are no less than m. Symmetrically, at least one-quarter of the elements in I are no greater than m, and so the `Select`s in steps 5 and 7 recurse on no more than three-quarters of the elements in I. Thus,

$T(|I|) \leq 2 \cdot scan(I) + T(|I|/cM) + T(3|I|/4)$; by induction, $T(|I|) = O(scan(|I|))$, assuming without loss of generality that $cM \geq 5$. □

3.2. *Relabeling*. Given forest F and edge set I, we construct the relabeling, $I' = RL(F, I)$ defined above, as follows:

1. Sort F by source vertex, v.
2. Sort I by second component.
3. Process F and I in tandem.
   (a) Let $\{s, h\} \in I$ be the current edge to be relabeled.
   (b) Scan F starting from the current edge until finding $(p(v), v)$ such that $v \geq h$.
   (c) If $v = h$, then add $\{s, p(v)\}$ to I''; otherwise, add $\{s, h\}$ to I''.
4. Repeat steps 2 and 3, relabeling first components of edges in I'' to construct I'.

As shown in Algorithm **CC**, relabeling can be used to effect pointer jumping, a technique widely applied in parallel graph algorithms [20]. Given a rooted forest $F = \{(p(v), v), \ldots\}$, *pointer jumping* produces a new rooted forest $F' = \{(p(p(v)), v) : (p(v), v) \in F\}$; i.e., each v of depth two or greater in F points in F' to its grandparent in F. (Define $p(v) = v$ if v is a root in F.) Our implementation of relabeling is similar to Chiang's [12] implementation of pointer jumping.

LEMMA 3.4. *Relabeling an edge list I by a forest F can be performed in $O(sort(|I|) + sort(|F|))$ I/Os in the FIO model.*

PROOF. Step 1 uses $O(sort(|F|))$ I/Os; step 2 uses $O(sort(|I|))$ I/Os; step 3 uses $O(scan(|F| + |I|))$ I/Os. □

3.3. *Contraction*. Define a *subcomponent* to be a collection of edges among vertices in the same connected component of G; subcomponents need not be maximal. Given a graph G and a list $C = \{C_1, C_2, \ldots\}$ of delineated subcomponents, the *contraction of G by C* is defined as the graph $G/C = G_{|C|}$, where $G_0 = G$, and for $i > 0$, $G_i = G_{i-1}/C_i$. That is, the vertices of each subcomponent in C are contracted into a supervertex.

Let I be the edge list of G, and assume that each C_i is presented as an edge list. (If each is input as a vertex list, the following procedure can be simplified.) We form an appropriate relabeling to I to effect the contraction, as follows:

1. For each $C_i = \{\{u_1, v_1\}, \ldots\}$:
   (a) $R_i \leftarrow \emptyset$.
   (b) Pick u_1 to be the canonical vertex.
   (c) For each $\{x, y\} \in C_i$, add (u_1, x) and (u_1, y) to relabeling R_i.
2. Apply relabeling $\bigcup_i R_i$ to I, yielding the contracted edge list I'.

For each C_i, one vertex, u_1, is picked to be the canonical vertex into which all others will be contracted. Step 1(c) adds an arc (u_1, v) to the relabeling forest for each vertex v in C_i. The result, R_i, is a star, rooted at u_1, with a leaf for each other vertex that appears in C_i. Each subcomponent, C_i, thus gets contracted into its canonical vertex in step 2.

LEMMA 3.5. *Contracting an edge list I by a list of delineated subcomponents $C = \{C_1, C_2, \ldots\}$ can be performed in $O(sort(|I|) + sort(\sum_i |C_i|))$ I/Os in the FIO model.*

PROOF. Step 1 uses $O(scan(|I|))$ I/Os and produces a relabeling forest of size $\sum_i |C_i|$. Applying Lemma 3.4 yields the final result. □

We can also save with each contracted edge the corresponding original edge. As we will see in Section 4.1, this facilitates later re-expansion.

3.4. *Deletion*. Given edge lists I and D, it is straightforward to construct $I' = I \backslash D$: simply sort I and D lexicographically, and process them in tandem to construct I' from the edges in I but not D.

Similarly, given a vertex list U, we can construct $I'' = \{\{u, v\} \in I : u \notin U \wedge v \notin U\}$. Sort U, and then sort I by first component; then process U and I in tandem, constructing list I' of edges in I whose first components are not in U. Then sort I' by second component, and process it in tandem with U, constructing list I'' of edges in I' whose second components are not in U. We abuse notation and write $I'' = I \backslash U$ when U is a set of vertices.

LEMMA 3.6. *Deleting a vertex or edge set of cardinality N from an edge set I can be performed in $O(sort(|I|) + sort(N))$ I/Os in the FIO model.*

4. Deterministic Algorithms.

We adapt Algorithm **CC** and use the transformations from Section 3 to produce efficient, deterministic, functional external algorithms for computing connected components, MSFs, BMSFs, and maximal matchings of undirected graphs.

4.1. *Connected Components, MSFs, and BMSFs*

LEMMA 4.1. *Algorithm* **CC** *runs in $O(sort(E) \log_2(E/M))$ I/Os in the FIO model.*

PROOF. Step 1 runs in $O(scan(E))$ I/Os. Sorting the edge list corresponding to $CC(G_1)$ by target vertex suffices to delineate $CC(G_1)$ by component, so step 3 runs in $O(sort(E))$ I/Os, by Lemma 3.5. Step 5 runs in $O(sort(E))$ I/Os, by Lemma 3.4. Each of steps 2 and 4 recurses on at most half the original edges. Thus, if $T(E)$ denotes the overall number of I/Os incurred on a graph of E edges, we have that $T(E) \le O(sort(E)) + 2T(E/2)$. We stop the recursion when a subproblem fits in internal memory, so $T(E) = O(sort(E) \log_2(E/M))$. □

THEOREM 4.2. *The delineated edge list of components of a graph can be computed in $O(sort(E) + (E/V) sort(V) \log_2(V/M))$ I/Os in the FIO model.*

PROOF. It suffices to compute a forest, F, of rooted stars corresponding to the connected components, because F can be used to delineate the original edge list in $O(sort(E))$

I/Os. We label each edge in E with its component in $O(sort(E))$ I/Os, by sorting E (by, say, first vertex) and F (by source vertex) and processing them in tandem to assign component labels to the edges. This creates a new labeled edge list, E''. We then sort E'' by label, creating the desired output E'.

Thus, if $E < V$, we are done, by Lemma 4.1. Otherwise, we apply the sparsification idea [16], partitioning E into $\lceil E/V \rceil$ lists of no more than V edges each. We compute the forest of rooted stars corresponding to the connected components of each sublist, in $O((E/V)sort(V)\log_2(V/M))$ I/Os overall by Lemma 4.1. We then iteratively merge pairs of forests, each time replacing two such forests by the forest of rooted stars corresponding to the connected components of their union. Again, this takes $O((E/V)sort(V)\log_2(V/M))$ I/Os overall by Lemma 4.1. That the resulting forest corresponds to the connected components of the original graph is shown by the same correctness proof of Algorithm **CC**. To see this, note that the correctness proof does not depend on the number of edges selected in step 1. The sparsification approach is equivalent to selecting V edges in step 1 if $E > V$, and selecting $E/2$ edges otherwise. □

Algorithm **CC** specializes the standard greedy algorithm for MSFs by assuming that all weights are uniform. We thus derive Algorithm **MSF**, denoting by $MSF(G)$ an MSF of G, presented as an edge list.

Algorithm **MSF**

```
1. Let $E_1$ be any lowest-cost half of the edges of $G$; i.e., every edge in $E \backslash E_1$
   has weight at least that of the edge of greatest weight in $E_1$. Let $G_1 =
   (V, E_1)$.
2. Compute $MSF(G_1)$ recursively.
3. Let $G' = G/MSF(G_1)$.
4. Compute $CC(G')$ recursively.
5. $MSF(G) = EX(MSF(G')) \cup MSF(G_1)$.
```

The transformation $EX(G')$ *expands* a contracted graph G' as follows. During the contraction that created G', each edge incident upon a supervertex corresponds to one original edge, which is included in the record for the new edge. $EX(G')$ replaces each contracted edge by its corresponding original edge. In Algorithm **MSF** the contraction in step 3 determines the original edges as follows. Consider contracting vertices x and y into supervertex z. For each vertex u incident upon either or both of x and y, a contracted edge $\{z, u\}$ is created. The edge $\{x, u\}$ or $\{y, u\}$ of smaller weight is chosen as the original edge to install in the record for $\{z, u\}$. In case of a tie, either may be chosen arbitrarily. The original edges are carried through the recursive calls; i.e., if $\{x, u\}$ or $\{y, u\}$ were themselves contracted edges, then, having chosen the edge of minimum weight between them, its corresponding original edge is installed as the original edge for $\{z, u\}$. Correctness of Algorithm **MSF** then follows by correctness of the standard greedy algorithm for MSFs [25].

THEOREM 4.3. *An MSF of a graph can be computed in $O(sort(E)+(E/V)sort(V)\log_2(V/M))$ I/Os in the FIO model.*

PROOF. The proof follows that of Theorem 4.2, using Lemma 3.3 to show that step 1 takes $O(sort(E))$ I/Os. The only complication is that the MSF returned in step 2 must be delineated by components in order to apply Lemma 3.5 to perform the contraction in step 3. To do so, we annotate the forest returned by $MSF(\cdot)$ with a corresponding forest of rooted stars representing the components. In step 5, we combine the two rooted-star forests representing the components of $MSF(G_1)$ and $MSF(G')$ as in step 5 of Algorithm **CC**. To produce the final result, we need only delineate the edge list of $MSF(G)$ as in the proof of Theorem 4.2. □

Algorithm **MSF** effects the standard greedy algorithm for computing a maximum-weight basis of a matroid [39] as applied to graphic matroids. Given appropriate external contraction and relabeling procedures, similar algorithms can be implemented for other matroids.

Recall now that a *bottleneck minimum spanning forest* (*BMSF*) minimizes the maximum weight of an edge. Since an MSF is also a BMSF, the I/O bound in Theorem 4.3 also applies to computing a BMSF.

We mention the BMSF problem because the internal BMSF algorithm of Camerini [10] runs in $O(E)$ time, faster than any known deterministic, comparison-based MSF algorithm. Camerini's algorithm is a one-sided recursion. If the lower-weighted half of the edges span the graph, they contain a BMSF, and so the upper half can be discarded. Otherwise, any BMSF contains an edge from the upper half, and so the lower half can be contracted.

Whether BMSFs can be computed externally more efficiently than MSFs is an open problem. If we could determine whether or not a subset $E' \subseteq E$ spans a graph in $g(E')$ I/Os, then we can use that procedure to limit the recursion to one-half of the edges of E, as in the classical BMSF algorithm. This would reduce the I/O complexity of finding a BMSF to $O(g(E) + sort(E))$ (*sort*(E) to perform the contraction).

4.2. *Maximal Matching.* Given a matching $\mathcal{M}$, denote by $V(\mathcal{M})$ the set of vertices matched by $\mathcal{M}$. We can use the framework of Algorithm **CC** to find a maximal matching of a graph $G = (V, E)$, denoted $MM(G)$.

Algorithm **MM**

```
1. Let $E_1$ be any non-empty, proper subset of edges of $G$; let $G_1 = (V, E_1)$.
2. Compute $MM(G_1)$ recursively.
3. Let $E' = E \backslash V(MM(G_1))$; let $G' = (V, E')$.
4. Compute $MM(G')$ recursively.
5. $MM(G) = MM(G') \cup MM(G_1)$.
```

THEOREM 4.4. *A maximal matching of a graph can be computed in* $O((E/V) sort(V) \log_2(V/M))$ *I/Os in the FIO model*.

PROOF. Deleting matched vertices in $MM(G_1)$ from G leaves a graph G' whose vertices are disjoint from $MM(G_1)$. Uniting a maximal matching of G' with $MM(G_1)$ produces a maximal matching of G, by the assumption of maximality of $MM(G_1)$, and thus Algorithm **MM** is correct.

The analysis of the I/O complexity follows that in the proof of Theorem 4.2. If $E < V$, then we choose $E/2$ edges in step 1; otherwise, we choose V edges. By Lemma 3.6, step 3 can be performed in $O(sort(N))$ I/Os, where N is the number of edges selected in step 1. □

To the best of our knowledge, the previous best result for external maximal matching is $O(sort(E)\log_2^3 V)$ I/Os, which can be obtained by applying PRAM simulation techniques to the algorithm of Israeli and Shiloach [19].

5. Randomized Algorithms.

We show how to externalize the randomized linear-time MSF algorithm of Karger et al. [21]. The result is randomized functional algorithms for connected components, MSFs, and BMSFs that incur $O(sort(E))$ I/Os with high probability. Similar approaches were suggested by Chiang et al. [13] and Mehlhorn [30].

We can also implement randomized functional external algorithms for maximal independent sets and maximal matchings, based on algorithms by Luby [28] and Yang et al. [40], respectively. Each uses $O(sort(E))$ I/Os with probability arbitrarily close to 1.

5.1. *Connected Components, MSFs, and BMSFs*.

Consider a weighted graph $G = (V, E)$. A *Borůvka step* [7], [21] selects and contracts the edge of minimum weight incident on each vertex. (Ties are broken lexicographically.) Borůvka steps are useful for two reasons. First, each Borůvka step at least halves the number of vertices in the graph. Second, it preserves the MSF of the contracted graph. More precisely, let G be a graph, let F be a subgraph of G contracted in a Borůvka step, and let G' be the resulting graph. Then the MSF of G is the MSF of G' plus the edges in F.

LEMMA 5.1. *If $B = O(N/\log^{(i)} N)$ for some fixed integer $i > 0$, then a Borůvka step can be performed in $O(sort(E))$ I/Os in the FIO model.*

PROOF. We implement the Borůvka step as follows. In $O(sort(E))$ I/Os we identify the minimum weight edge incident on each vertex: in turn, sort by first and second components of each edge and scan to select the minimum weight edge. Call the set of edges chosen F.

Since F is a forest, it belongs to a family of sparse graphs closed under edge contraction. Therefore, we can find its connected components in $O(sort(V))$ I/Os using a result of Chiang et al. [13]. (As noted in Section 2.1, the techniques of Chiang et al. are functional. The assumption that $B = O(N/\log^{(i)} N)$ for some fixed integer $i > 0$ underlies the list-ranking results used by Chiang et al. to find the components.) Labeling each edge in F by its connected component allows us to rearrange the edges into a list delineated by component, in $O(sort(E))$ I/Os. We can then contract E by F in $O(sort(E))$ I/Os, by Lemma 3.5. □

Arge [4] shows how to eliminate the restriction on B, although the result is not functional. Kumar and Schwabe [26] also effect a Borůvka step without this assumption, again, though, not in a functional manner.

Karger et al. [21] combine Borůvka steps with a random selection technique that also at least halves the number of edges, resulting in a linear-time randomized MSF algorithm, which we can directly externalize. Their algorithm proceeds as follows:

1. Perform two Borůvka steps, which reduces the number of vertices by at least a factor of four. Call the contracted graph G'.
2. Choose a subgraph H of G' by selecting each edge independently with probability $1/2$.
3. Apply the algorithm recursively to find the MSF F of H.
4. Delete from G' each edge $\{u, v\}$ such that (1) there is a path, $P(u, v)$, from u to v in F and (2) the weight of $\{u, v\}$ exceeds that of the maximum-weight edge on $P(u, v)$. Call the resulting graph G''.
5. Apply the algorithm recursively to G'', yielding MSF F'.
6. Return the edges contracted in step 1 together with those in F'.

THEOREM 5.2. *If $B = O(N/\log^{(i)} N)$ for some fixed integer $i > 0$, then an MSF of a graph can be computed in $O(sort(E))$ I/Os with probability $1 - e^{-\Omega(E)}$ in the FIO model.*

PROOF. Karger et al. [21] prove the correctness of the above algorithm. We can perform step 1 in $O(sort(E))$ I/Os, by Lemma 5.1. Step 3 recurses on a subgraph with (expected) half the original edges. Step 4 can be accomplished via an MSF verification step [21], which can be performed in $O((E/V) sort(V)) = O(sort(E))$ I/Os [13]. (The restriction on B applies to the MSF verification step as well as the Borůvka step. Again, the techniques of Chiang et al. [13] are functional.) Karger et al. [21] show that G'' has, on expectation, about $V/4$ vertices and $V/8$ edges, where V is the original vertex set. Using these facts in the proof of Theorem 4.3 of Karger et al. [21] completes the proof. □

COROLLARY 5.3. *If $B = O(N/\log^{(i)} N)$ for some fixed integer $i > 0$, then the delineated edge list of components of a graph can be computed in $O(sort(E))$ I/Os with probability $1 - e^{-\Omega(E)}$ in the FIO model.*

PROOF. The MSF of the graph yields its connected components. □

COROLLARY 5.4. *If $B = O(N/\log^{(i)} N)$ for some fixed integer $i > 0$, then a BMSF of a graph can be computed in $O(sort(E))$ I/Os with probability $1 - e^{-\Omega(E)}$ in the FIO model.*

PROOF. Any MSF is also a BMSF, so we can apply Theorem 5.2. □

5.2. *Maximal Independent Sets and Matchings*. Recall Luby's randomized maximal independent set algorithm [28]:

1. $I \leftarrow \emptyset$; $V' \leftarrow V$; $E' \leftarrow E$.
2. Set $d(v)$ to the degree of vertex v, $\forall v \in V'$.

3. Mark v with probability $1/2d(v)$ if $d(v) > 0$ and probability 1 if $d(v) = 0$, $\forall v \in V'$.
4. For each $\{u, v\} \in E'$ such that both u and v are marked, if $d(u) \leq d(v)$, then unmark u; otherwise unmark v.
5. $I' \leftarrow \{v \in V' : v \text{ is marked}\}$; $I \leftarrow I \cup I'$; $Y \leftarrow I' \cup \{v \in V' : \exists \{u, v\} \in E' \ni u \in I'\}$.
6. $V' \leftarrow V' \backslash Y$; $E' \leftarrow E' \backslash Y$.
7. If $V' \neq \emptyset$, repeat from step 2.

THEOREM 5.5. *For any fixed ε, a maximal independent set of a graph can be computed in $O(sort(E))$ I/Os with probability at least $1 - \varepsilon$ in the FIO model.*

PROOF. Luby [28] shows that I is a maximal independent set at the completion of the algorithm. Each of steps 2–6 can be performed in $O(sort(E))$ I/Os. Step 2 sorts E (in turn by each vertex) and then scans E to compute the vertex degrees, producing a list D containing $d(v)$ for each vertex v. Step 3 scans D to compute a list X of marked vertices. Step 4 scans X, D, and E in tandem to apply the marks and degrees to the vertices. (This requires a second sort of E, by the second vertex component, and scan to mark both components of E.) Then E is scanned to produce a list X' of vertices to be unmarked. $X \leftarrow X \backslash X'$ can be computed by sorting X' and scanning it in tandem with X. Step 5 sorts I' and scans it in tandem with E' to produce Y. Step 6 takes $O(sort(E))$ I/Os, by Lemma 3.6.

Let $T(E)$ be the number of I/Os used by the algorithm. Luby [28, Theorem 1] shows that step 6 reduces the number of edges by an expected factor of at least one-eighth. Theorem 1.4 of Karp [22] thus shows that

$$\Pr[T(E) \geq (8 + w)sort(E)] \leq \left(\tfrac{7}{8}\right)^w$$

for every positive integer w. □

Maximal matchings of a graph G correspond to maximal independent sets of its associated line graph, $L(G)$ [23]. $L(G)$ can have as many as E^2 edges, however, and so Theorem 5.5 would give an I/O bound of $O(sort(E^2))$ for finding a maximal independent set of $L(G)$ (and thus a maximal matching of G). We therefore implement randomized external maximal matching directly, using the following algorithm of Yang et al. [40] (which is based on a similar algorithm by Israeli and Itai [18]):

1. $\mathcal{M} \leftarrow \emptyset$.
2. Set the label of v to 0 with probability 1/2 and to 1 with probability 1/2, $\forall v \in V$.
3. For each $u \in V$ such that u is labeled 1, pick any adjacent v such that v is labeled 0. (If u has no adjacent 0-labeled vertex, then u makes no choice.) Let E' be the resulting set of $\{u, v\}$ edges.
4. Let V' be the 0-labeled vertices among the edges in E'. For each $v \in V'$, pick any one incident edge $\{v, w\} \in E'$. (Note that w is labeled 1.) Let E'' be the resulting set of $\{v, w\}$ edges.
5. $\mathcal{M} \leftarrow \mathcal{M} \cup E''$.
6. $E \leftarrow E \backslash E''$.
7. If $E \neq \emptyset$, repeat from step 2.

THEOREM 5.6. *For any fixed ε, a maximal matching of a graph can be computed in $O(sort(E))$ I/Os with probability at least $1 - \varepsilon$ in the FIO model.*

PROOF. Step 3 induces a set of stars, E', each with a root labeled 0 and leaves labeled 1. Step 4 picks one edge from each star. Thus E'' is a matching, and, at termination, M is a maximal matching, by steps 5 and 6.

Step 2 takes $O(scan(V))$ I/Os; we can assume that V is sorted by vertex at a one-time cost of $O(sort(V))$ I/Os. Each of steps 3–6 can be implemented in $O(sort(E))$ I/Os. Step 3 sorts E in turn by each edge component and traverses it in tandem with V to assign vertex labels to the edge components. A second scan of E (sorted by the first, or 1-labeled, component) suffices to produce E'. Step 4 then sorts E' by the second component (0-labeled vertex) and scans the sorted list to form E''. Step 6 takes $O(sort(E))$ I/Os, by Lemma 3.6.

Let $T(E)$ be the number of I/Os used by the algorithm. Yang et al. [40, Lemma 2] show that step 6 reduces the number of edges by an expected factor of at least $\eta = \frac{1}{4}\left[1 - e^{-1/3}\right]$. Theorem 1.4 of Karp [22] thus shows that

$$\Pr\left[T(E) \geq \left(\frac{1}{\eta} + w\right) sort(E)\right] \leq (1-\eta)^w$$

for every positive integer w. □

6. Semi-External Problems.

We now consider *semi-external* graph problems, when $V \leq M$ but $E > M$. These cases often have practical applications, e.g., in graphs induced by monitoring long-term traffic patterns among relatively few nodes in a network. For example, a graph induced by telephone calls in the AT&T network has about 250 million vertices but can have billions of edges (200–300 million edges for each day of traffic). The ability to maintain in memory information about the vertices often simplifies the problems.

For example, if $V \leq cM$ for some suitable constant $0 < c < 1$, we can compute the forest of rooted stars corresponding to the connected components of a graph in one scan, using disjoint set union [34] to maintain a forest internally. To compute MSFs, we similarly maintain a forest internally, first sorting the edge list by weight. (This is Kruskal's algorithm [25].) We can even eliminate the sort and compute MSFs in *scan*(E) I/Os if we use dynamic trees [33] to maintain the internal forest. For each edge, we delete the maximum weight edge on any cycle created. The total internal computation time then becomes $O(E \log V)$.

Semi-external BMSFs are similarly simplified, because we can check internally if an edge subset spans a graph. Semi-external maximal matching is possible in one edge-list scan, simply by maintaining a matching internally.

This semi-external approach differs from that used by Chiang et al. [13] to implement external depth-first search (DFS). Chiang et al. maintain a data structure internally on the vertices, but they do not assume that $V = O(M)$ and thus that the data structure fits entirely in main memory. Once it grows too large, information is compacted, and a new phase of the algorithm is started. This results in an $O(\lceil V/M \rceil scan(E) + V)$-I/O algorithm for DFS. In contrast, our semi-external algorithms maintain the entire vertex data structure in main memory at all times.

Having computed the intermediate forest data structure without sorting, consider the final step of arranging the components in a delineated edge list. We can label the edges by their components in one scan, and we can then sort the edge list to arrange edges contiguously by component, in *sort*(E) I/Os. The sorting bound is pessimistic, however, if the number of components is small. Below we give an algorithm to sort N records with keys in the integral range $[1, K]$ in $O(scan(N)\log_{M/B} K)$ I/Os. If there are K distinct keys from an arbitrary universe, we get the same I/O bound with high probability for the relaxed problem of grouping the records so that records with distinct keys appear contiguously. These bounds are tight [29]. Therefore, if $V \leq cM$ we can compute connected components on a graph $G = (V, E)$ in $O(scan(E)\log_{M/B} C(G))$ I/Os, where $C(G) \leq V$ is the number of components of G. In many applications, $C(G) \ll V$, so this approach significantly improves upon sorting. In the telephone-graph example above, the number of connected components generated by a typical day of data is less than 10 million; the number of components containing more than 10 vertices, in fact, is less than 10,000. These figures become even smaller as more edges are processed.

6.1. *Sorting with Duplicates*. We externalize a version of distribution sort [24] to sort N items with keys in the integral range $[1, K]$. The basic idea is straightforward: the items are partitioned into M/B buckets, each of which contains items in a distinct range of KB/M keys. Knowing K in advance (or calculating it in one scan) allows each key to be assigned to its corresponding bucket. Each block in main memory is assigned to hold items in a single bucket. As the input is read from disk, each item is assigned to its appropriate memory block. When a memory block becomes full, it is written to an appropriate place on disk to maintain the items from the corresponding bucket. After processing the input, the buckets are sorted recursively. (We ignore that one extra block of memory is required to allow scanning of the input; this does not affect the bounds.)

To make the approach functional, care must be taken in the output. We assume that a disk block can be addressed with $O(1)$ memory cells. We can thus chain disk blocks into a linked list without affecting the asymptotic space usage. Each memory block maintains a pointer (initially null) to the first block in the linked list corresponding to its bucket being output. When a memory block becomes full, it is output to a new disk block, which is prepended to the list, and the pointer in the memory block is reset to point to the new disk block. (Prepending does not require updating successive blocks in the chain.)

After the input has been scanned, some memory blocks will be non-empty. Those with non-null pointers are emptied in the above fashion, i.e., prepended to their respective bucket chains. The remaining memory blocks were never emptied during the scan, because they never became full. These can be partitioned into contiguous ranges of keys, each range maximally fitting in order between two chains of buckets that have been written to disk. The final output is produced by scanning through the chains on disk, in order, interspersing the ranges of keys remaining in main memory in order, to produce a bucketed list requiring only $\lceil N/B \rceil$ disk blocks.

THEOREM 6.1. *Sorting N records with keys in the integral range* $[1, K]$ *takes* $\Theta(\mathit{scan}(N) \log_{M/B} K)$ *I/Os in the FIO model*.

PROOF. The upper bound follows from the above discussion. Every chain written to disk contains at least one full disk block, which can be charged for the (at most one) partial block prepended in the clean-up phase after the input has been scanned. Each range containing keys never written to disk might result in only a partial disk block being written in the clean-up phase, but each such range can be amortized against the preceding chain, which contains at least one full disk block. (The first range might be at the head of the output, but this accounts for at most one partial disk block.)

The lower bound comes from Matias et al. [29], who improve the previous lower bound based on duplicate elimination due to Arge [5] in the case when $M < B^2$. □

Subsequently, Matias et al. [29] have shown how the above algorithm can be implemented in place. While not functional, in-place sorting routines are important when minimizing work space is critical.

COROLLARY 6.2. *If* $V \leq cM$ *for some suitable constant* c, *the delineated edge list of components of a graph can be computed in* $O(\mathit{scan}(E) \log_{M/B} C(G))$ *I/Os in the FIO model*, *where* $C(G)$ *is the number of components of the graph*.

PROOF. By the discussion at the beginning of Section 6, we can label each edge by its component in $O(\mathit{scan}(E))$ I/Os. Theorem 6.1 completes the proof. □

Note that sorting solves the *proximate neighbors* problem [13]: given N records on $N/2$ distinct keys, such that each key is assigned to exactly two records, permute the records so that records with identical keys reside in the same disk block. Chiang et al. [13] show a lower bound of $\Omega(\min\{N, \mathit{sort}(N)\})$ I/Os to solve proximate neighbors in the single-disk I/O model. Theorem 6.1 does not violate this bound, since in the proximate neighbors problem, $K = N/2$.

6.2. *Grouping with Arbitrary Keys*. If the N records have K distinct keys that span an arbitrary range, we can still use the above scheme to *group* them so that records with identical keys appear contiguously in the output; records with different keys, however, do not necessarily obey the total order.

During each recursive phase, we pick a hash function, h, independently and uniformly at random from a family of universal hash functions [11] that map the K keys to the range $[1, \lfloor M/B \rfloor]$. When scanning the input, we use h to assign keys to buckets.

Equal keys are always hashed to the same bucket. When a bucket contains at most M/B distinct keys, we can group its records in one additional scan. Linking the records in the buckets output in the first phase, denoted by T, in which each bucket has no more than M/B distinct keys, therefore, produces the desired grouped output.

THEOREM 6.3. *With probability at least $1 - 1/K^c$ for any fixed constant c, $O(scan(N) \log_{M/B} K)$ I/Os suffice to group N items with K distinct keys from an arbitrary universe in the FIO model.*

PROOF. The properties of universal hashing [11] show that, in each recursive phase, the expected number of distinct keys assigned to any bucket is a fraction M/B of the number of keys being processed. Theorem 1.1 of Karp [22] thus shows that

$$\Pr[T \geq \lfloor \log_{M/B} K \rfloor + w + 1] \leq \frac{K}{(M/B)^{\lfloor \log_{M/B} K \rfloor + w}}$$

for any positive integer w. In particular, let $w = c\lceil \log_{M/B} K \rceil + 1$. □

7. Conclusion.

Our functional approach produces external graph algorithms that compete with the I/O performance of the best previous algorithms, are simple to describe and implement, and are conducive to standard checkpointing and programming language optimization tools. An interesting open problem is to devise incremental and dynamic algorithms for external graph problems. The data-structural approach of Arge [4] and Kumar and Schwabe [26] holds promise for this area.

It remains open to determine whether or not testing a graph for connectedness (i.e., testing if the number of connected components is one) is as hard externally as computing its connected components. An easier connectivity test could lead to an improved external BMSF algorithm.

Finally, our procedure to implement a Borůvka step is complicated by the fact that we can only contract a list of E edges in $O(sort(E))$ I/Os if they are delineated by component. Kumar and Schwabe show how to contract an arbitrary set of E edges in $O(sort(E))$ I/Os, but their procedure is not functional. How to contract an arbitrary edge list functionally in constant sorts remains open.

Acknowledgments. We thank Lars Arge, Ken Church, Kathleen Fisher, and Anne Rogers for discussions about the functional approach; David Karger and Kurt Mehlhorn for discussions about the randomized algorithms; and David Johnson and Haim Kaplan for useful feedback. We also thank an anonymous referee for extensive, constructive comments.

References

[1] J. Abello and J. S. Vitter, editors. *External Memory Algorithms*, volume 50 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, Providence, RI, 1999.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(8):1116–27, 1988.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[4] L. Arge. The buffer tree: a new technique for optimal I/O algorithms. In *Proc*. 4*th Workshop on Algorithms and Data Structures*, volume 955 of Lecture Notes in Computer Science, pages 334–45. Springer-Verlag, Berlin, 1995.

[5] L. Arge. Efficient External-Memory Data Structures and Applications. Ph.D. thesis, Dept. of Computer Science, University of Aarhus, 1996.

[6] L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of Geographic Information Systems*, volume 1340 of Lecture Notes in Computer Science, pages 213–54. Springer-Verlag, Berlin, 1997.

[7] O. Borůvka. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti v Brně* (*Acta Societ. Science. Natur. Moravicae*), 3:37–58, 1926.

[8] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc*. 6*th Scandinavian Workshop on Algorithm Theory*, volume 1432 of Lecture Notes in Computer Science, pages 107–18. Springer-Verlag, Berlin, 1998.

[9] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc*. 11*th ACM–SIAM Symposium on Discrete Algorithms*, pages 859–60, 2000.

[10] P. M. Camerini. The min-max spanning tree problem and some extensions. *Information Processing Letters*, 7:10–14, 1978.

[11] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–54, 1979.

[12] Y.-J. Chiang. Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results. Ph.D. thesis, Dept. of Computer Science, Brown University, 1995.

[13] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc*. 6*th ACM–SIAM Symposium on Discrete Algorithms*, pages 139–49, 1995.

[14] F. Y. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–65, 1982.

[15] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[16] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–96, 1997.

[17] R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):354–62, 1999.

[18] A. Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22:77–80, 1986.

[19] A. Israeli and Y. Shiloach. An improved parallel algorithm for maximal matching. *Information Processing Letters*, 22:57–60, 1986.

[20] J. Ja'Ja. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.

[21] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–28, 1995.

[22] R. M. Karp. Probabilistic recurrence relations. *Journal of the ACM*, 41(6):1136–50, 1994.

[23] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. *Journal of the ACM*, 32(4):762–73, 1985.

[24] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 1973.

[25] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:53–57, 1956.

[26] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc*. 8*th IEEE Symposium on Parallel and Distributed Processing*, pages 169–76, 1996.

[27] M. J. Litzkow and M. Livny. Making workstations a friendly environment for batch jobs. In *Proc*. 3*rd Workshop on Workstation Operating Systems*, pages 62–67, April 1992.

[28] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–53, 1986.

[29] Y. Matias, E. Segal, and J. S. Vitter. Efficient bundle sorting. In *Proc*. 11*th ACM–SIAM Symposium on Discrete Algorithms*, pages 839–48, 2000.

[30] K. Mehlhorn. Personal communication. `http://www.mpi-sb.mpg.de/~crauser/courses.html`, 1998.

[31] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc.* 10*th ACM–SIAM Symposium on Discrete Algorithms*, pages 687–94, 1999.

[32] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: transparent checkpointing under UNIX. In *Proc. USENIX Winter* 1995 *Technical Conference*, pages 213–23, 1995.

[33] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–91, 1983.

[34] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–81, 1984.

[35] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence*, 3(2–4):331–60, 1991.

[36] J. S. Vitter. External memory algorithms. In *Proc.* 17*th ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems*, pages 119–28, 1998.

[37] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–48, 1990.

[38] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *Proc.* 25*th IEEE International Symposium on Fault Tolerant Computing*, pages 22–31, 1995.

[39] D. J. A. Welsh. *Matroid Theory*. Academic Press, New York, 1976.

[40] S. B. Yang, S. K. Dhall, and S. Lakshmivarahan. Simple randomized parallel algorithms for finding a maximal matching in an undirected graph. In *Proc. IEEE SOUTHEASTCON* ’91, volume 1, pages 579–81, 1991.