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

Algorithmica

A Functional Approach to External Graph Algorithms[1]

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

Presenter: Tao Sun

Background

Motivation:

Classical algorithms often do not scale when data exceed main memory limits. Graph algorithms for RAMs are poorly suited for direct extension to external memory, because of the lack of locality in graph data. Current approaches do not address the I/O implications of graph traversal very well. Therefore, it is interesting to develop algorithms that can process data in external storage.

Idea:

In this paper, the authors proposed a new divide-and-conquer functional approach for designing external graph algorithms. They applied their methodology 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. Their I/O bounds compete with these previous approaches

I/O model of complexity

N = number of items in the instance,
M = number of items that can fit in main memory,
B = number of items per disk block.

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 a graph, we define V to be the number of vertices, E to be the number of edges, and $N = V + E$.

- Their external algorithms rely on scanning and sorting as primitives:
- $sort(N) = O((N/B)\log_{M/B}(N/B))$ --- the number of I/Os needed to sort N items
- $scan(N) = \lceil N/B \rceil$ -- the number of I/Os needed to transfer N contiguous items between disk and internal memory.

Problem definitions

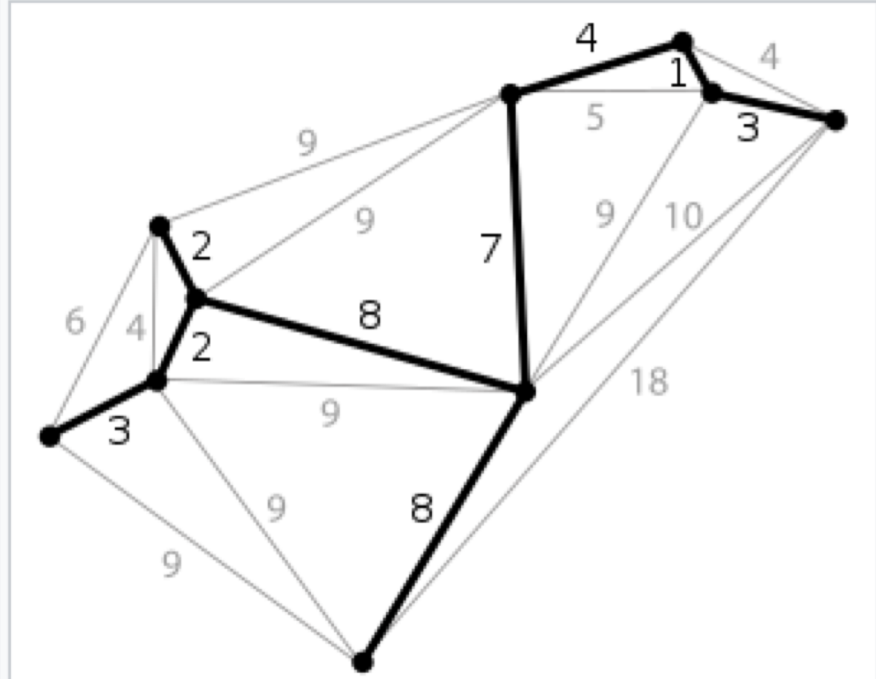
Connected components. A *connected component* is the edge set induced by a maximal set of vertices such that each pair of vertices is connected by a path in G. The output is a *delineated list* of edges, $\{C_1, C_2, \ldots, C_k\}$, where k is the number of components. Each C_i is the list of edges in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



A planar graph and its minimum spanning tree. Each edge is labeled with its weight, which here is roughly proportional to its length.

I/O bounds results

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

Problem	Deterministic	Randomized	
	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

Previous methods

PRAM simulation

Simulate CRCW PRAM using 1 processor and an external disk

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.

1. 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.
2. 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 $r(i) = A[d(i)]$
3. List R is then sorted by processor number, and the sorted R and T are scanned in tandem
4. 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
5. 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;
6. T' contains the updated processor states.

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.

no algorithm based on the simulation has been implemented.

Buffering data structure

Different buffer tree structures were designed to support sequences of insert, delete and deltetemin operations

Each node maintains a large 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.

Such data structures excel in computational geometry applications, they are hard to apply to external graph algorithms

Functional graph transformation

The authors proposed a divide-and-conquer mechanism to transform the graph. For a connected component :

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.

Contraction

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 $\{z,z\}$.
Consider a set, E', of vertex pairs, and let G/E' denote the result of contracting all vertex pairs in E . 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

Relabeling

Given a rooted forest F as an unordered sequence of oriented tree edges $\{(p(v),v),...\}$, 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.

We generalize the above into a purely functional approach to design external graph algorithms. Formally, let $f_P(G)$ 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 $f_P(G_1)$ to create a new subgraph, G_2. Let T_2 be a transformation that maps the solutions $f_P(G_1)$ and $f_P(G_2)$, 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. $f_{\mathcal{P}}(G) = T_2(G, G_1, G_2, f_{\mathcal{P}}(G_1), f_{\mathcal{P}}(G_2))$.

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.

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 $\texttt{Select}(I_3, k - |I_1| - |I_2|)$.

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'.

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'.

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.

Deterministic Algorithms

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

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. $MSF(G) = EX(MSF(G')) \cup MSF(G_1)$.

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.*

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. $MM(G) = MM(G') \cup MM(G_1)$.

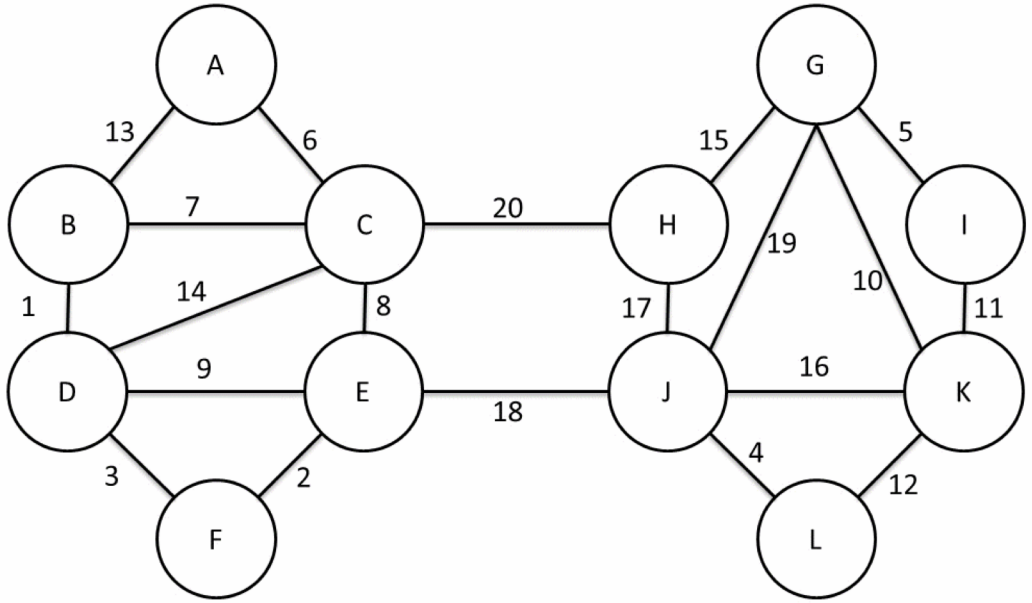
THEOREM 4.4. *A maximal matching of a graph can be computed in* $O((E/V)sort(V) \log_2(V/M))$ *I/Os in the FIO model.*

Randomized Algorithms

Borůvka's algorithm is a greedy algorithm for finding a minimum spanning tree in a graph, or a minimum spanning forest in the case of a graph that is not connected.

1. Begins by finding the minimum-weight edge incident to each vertex of the graph, and adding all of those edges to the forest.
2. Then, it repeats a similar process of finding the minimum-weight edge from each tree constructed so far to a different tree, and adding all of those edges to the forest.

Each repetition of this process reduces the number of trees, within each connected component of the graph, to at most half of this former value, so after logarithmically many repetitions the process finishes. When it does, the set of edges it has added forms the minimum spanning forest.



Randomized Algorithms

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.

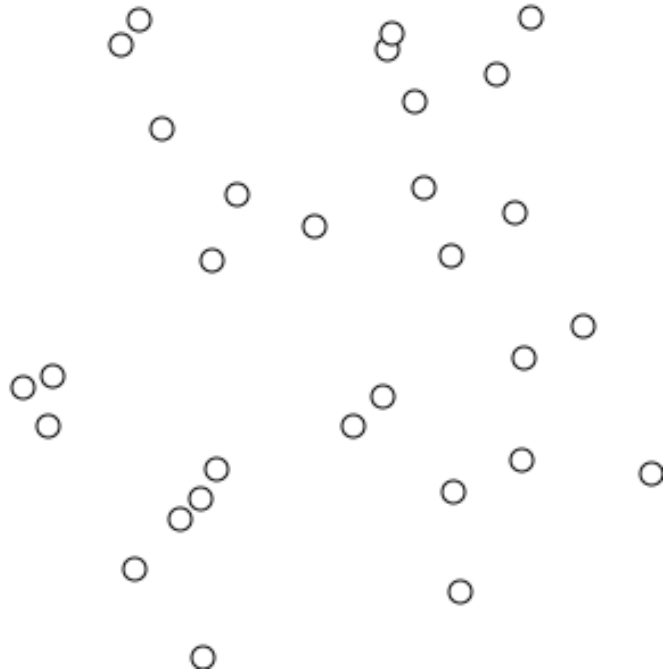
LEMMA 5.1. *If* $B = O(N/\log^{(i)} N)$ *for some fixed integer* $i > 0$, *then a Borůvka step can be performed in* $O(sort(E))$ *I/Os in the FIO model.*

Semi-External Problems

We now consider semi-external graph problems, when V ≤ 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. The ability to maintain in memory information about the vertices often simplifies the problems.

Kruskal's algorithm finds a minimum_spanning_forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum_spanning_tree.

- create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
- create a set *S* containing all the edges in the graph
- while *S* is non-empty and *F* is not yet spanning
  - remove an edge with minimum weight from *S*
  - if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree



At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree.

Semi-External 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 $O(E \log V)$.

Contributions

The authors proposed a functional approach to produce external graph algorithms

- Equivalent with the I/O performance of the best previous algorithms
- Simple to describe and implement
- Conducive to standard checkpointing and programming language optimization tools.