A Functional Approach to External Graph Algorithms

J. Abello, A. L. Buchsbaum, and J. R. Westbrook

Presenter: Nguyen Le

Motivation

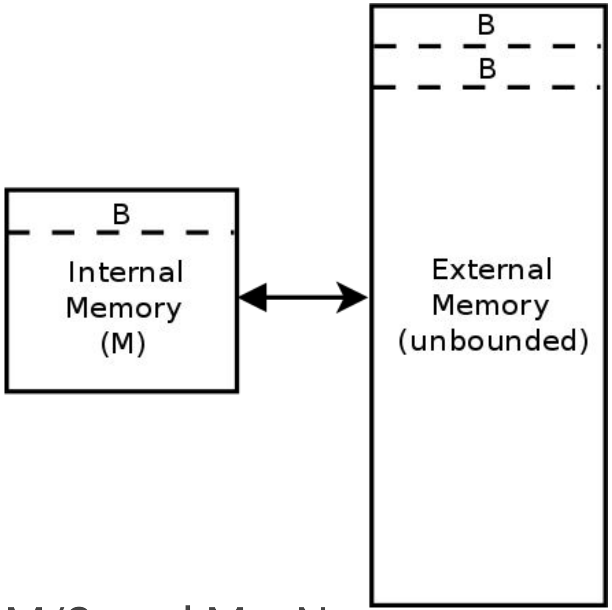
- Current algorithms do not completely address the I/O implications of graph traversal
- This paper producing algorithms that are purely **functional**
  - Functions applied to input data and producing output data
  - Information, once written, remains unchanged
- Allows standard **checkpointing** techniques
- Amenable to general purpose programming language transformations - reduce running time
- New divide-and-conquer approach
- Divise external algorithms for graph problems

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.



Typical computer server: $M \approx 10^9$ and $B \approx 10^3$; $1 < B < M/2$, and $M < N$.

Assume that $B = O(N/\log^{(i)} N)$ for some fixed integer $i > 0$

Definitions for graph

V = number of vertices

E = number of edges

N = V + E = number of items in the instance

$sort(N) = \Theta((N/B)\log_{M/B}(N/B))$,

$scan(N) = \lceil N/B \rceil$

Goal: replace N by N/B and $\log_2$ by $\log_{M/B}$

Problems

- Connected components
  - Maximal set of vertices such that each pair of vertices is connected by a path
- Minimum spanning forests
  - Spanning forest that minimizes the sum of the weights of the edges
- Bottleneck minimum spanning forests
  - Spanning forest that minimizes the weight of the maximum edge
- Maximal matching
  - Maximal set of edges such that no two edges share a common vertex
- Maximal independent set
  - Maximal set of vertices such that no two vertices are adjacent

Previous approaches

- PRAM Simulation
  - Simulate a CRCW PRAM algorithm using one processor and an external disk
  - Not practical - No algorithm based on the simulation has been implemented
  - Typically used to prove the existence of an external memory algorithm of a given I/O complexity
- Buffering Data Structures
  - *Buffer trees*, which support sequences of insert, delete, and deletemin operations on N elements
  - Hard to apply external graph algorithms
  - Data structure is not functional

Functional Graph Transformations

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. $f_{\mathcal{P}}(G) = T_2(G, G_1, G_2, f_{\mathcal{P}}(G_1), f_{\mathcal{P}}(G_2))$.

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. $CC(G) = CC(G') \cup RL(CC(G'), CC(G_1))$.

- Functional if S, T_1, and T_2 can be implemented without side effects on their arguments
- **Selection, relabeling, contraction, and (vertex and edge) deletion** can be implemented functionally

Selection

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

Relabeling

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'.

Contraction

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.

Vertex/ Edge Deletion

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.

CC, MM, MSF

Framework

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 **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. $CC(G) = CC(G') \cup RL(CC(G'), CC(G_1))$.

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

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

BMSF (Bottleneck MSF)

- If the lower-weighted half of the edges span the graph, they contain a BMSF - discard lower half
- Otherwise, any BMSF contains an edge from the upper half - discard upper half
- Open problem whether BMSFs can be computed externally more efficiently than MSFs

Randomized Algorithms

Boruvka Step - *O(sort(E))* I/Os

- Identify (and contract) the minimum weight edge incident on each vertex
- Sort by first and second components of each edge. Scan to select minimum weight edge

- Halves number of vertices
- Preserves the MSF of the contracted graph

Randomized Algorithms

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'.

Semi-external Problems

- $V \leq M$ but $E > M$. Example: monitoring long-term traffic, telephone calls
- Maintain in memory information about the V simplifies the problems
- MSF - $O(E \log V)$- using dynamic tree to maintain the internal forest
- BMSFs - check internally if an edge subset spans a graph

Results

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

Open problems

- Parallel disks
- Devise incremental and dynamic algorithms for external graph problems
- Determine whether or not testing a graph for connectedness
  - Easier testing -> Improved BMSFs