Single Source Shortest Paths (SSSP)

Directed graph $G = (V, E)$

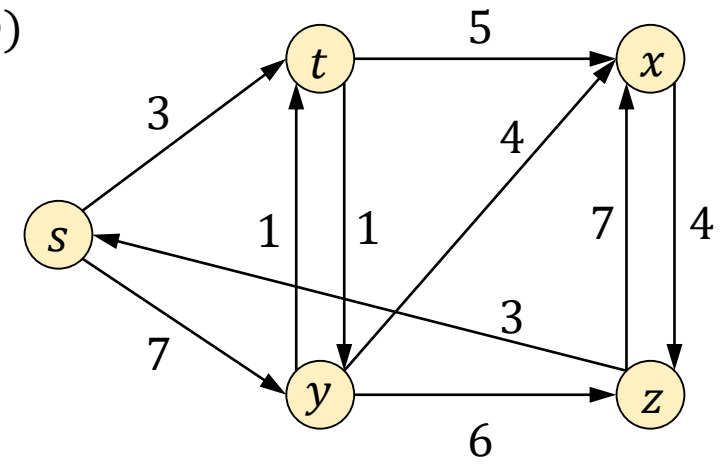
Edge weights $w : E \to \mathbf{R}$

Shortest path from s to v :

Path $p = (v_0, v_1, \ldots, v_k)$ of minimum weight $w(p)$

where $v_0 = s$, $v_k = v$ and

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_k)$$



SSSP problem

Compute a shortest path from source s to all vertices v

$d(v)$ = distance (i.e., shortest path weight) from s to v

Single Source Shortest Paths (SSSP)

Directed graph $G = (V, E)$

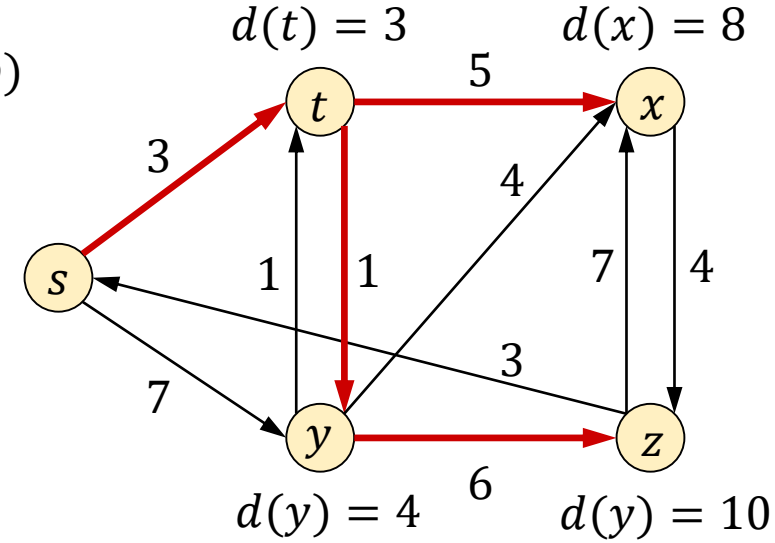
Edge weights $w : E \to \mathbf{R}$

Shortest path from s to v :

Path $p = (v_0, v_1, \ldots, v_k)$ of minimum weight $w(p)$

where $v_0 = s$, $v_k = v$ and

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_k)$$



SSSP problem

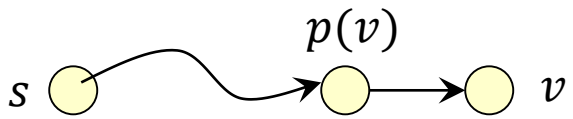
Compute a shortest path from source s to all vertices v

$d(v)$ = distance (i.e., shortest path weight) from s to v

Single Source Shortest Paths (SSSP)

Parent of a vertex

$p(v) =$ vertex just before v on the shortest path from s



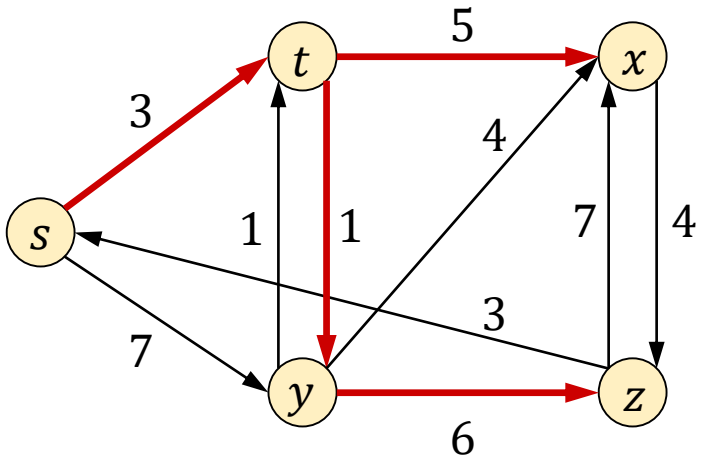
Shortest paths tree

Formed by the edges $(p(v), v)$

$p(s) =$ -

$p(t) = s$ $\quad$ $p(y) = t$

$p(x) = t$ $\quad$ $p(z) = y$



Single Source Shortest Paths (SSSP)

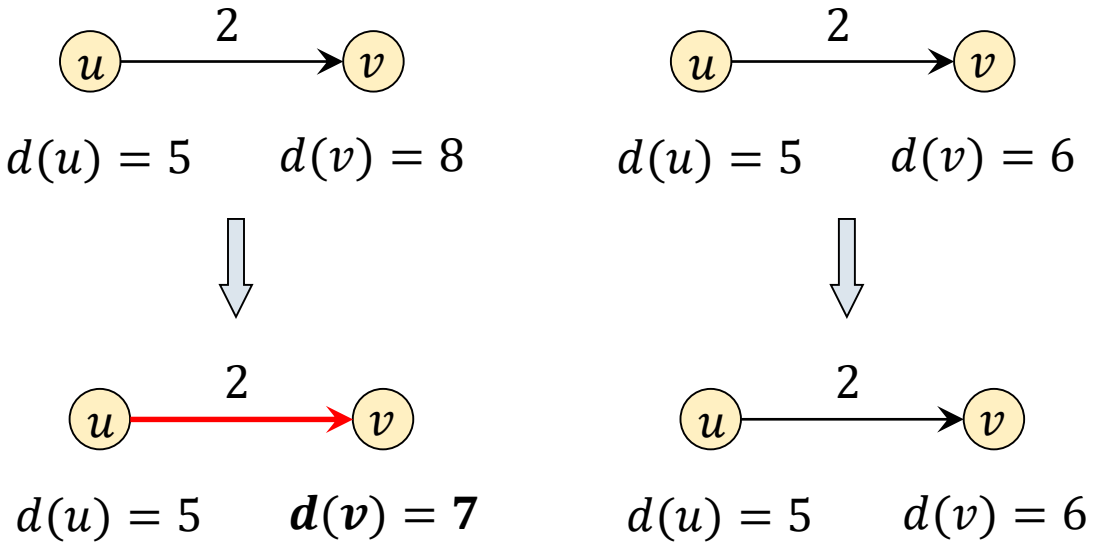
Temporary distances

$d(v)$ = upper bound for the weight of the shortest path from s to v

Initialize $p(v) \leftarrow \text{null}, d(v) \leftarrow \infty$ for all $v \neq s$

$p(s) \leftarrow \text{null}, d(s) \leftarrow 0$

Edge relaxation



relax(u, v)

if $d(v) > d(u) + w(u, v)$

then {

$d(v) \leftarrow d(u) + w(u, v)$

$p(v) \leftarrow u$

}

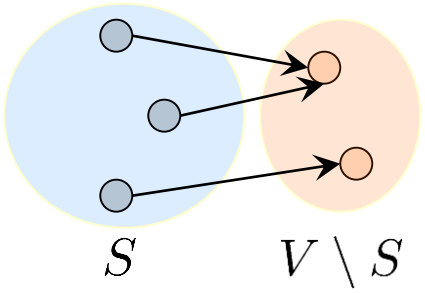
Single Source Shortest Paths (SSSP)

Dijkstra's Algorithm

Used when edge weights are non-negative $w(u, v) \geq 0, \ \forall \ (u, v) \in E$

It maintains a set of vertices $S \subseteq V$ for which a shortest path has been computed, i.e., the value of $d(v)$ is the exact weight of the shortest path to v.

Each iteration selects a vertex $u \in V \backslash S$ with minimum distance $d(u)$.
Then we set $S \leftarrow S \cup \{u\}$ and relax all edges (u, w)



To find u with $\min d(u)$: Use a priority queue Q with keys $d(u), \forall \ u \in V \setminus S$

Single Source Shortest Paths (SSSP)

Dijkstra's Algorithm

Initialization

$p(v) \leftarrow \text{null}, d(v) \leftarrow \infty$ for all $v \neq s$

$p(s) \leftarrow \text{null}, d(s) \leftarrow 0$

set $S \leftarrow \emptyset$

insert all vertices v into priority queue Q with key $d(v)$

Main Loop

while Q is not empty {
    $u \leftarrow \text{Q.delMin}()$
    $S \leftarrow S \cup \{u\}$
    for all edges (u, v) {
        $\text{relax}(u, v)$
    }
}

Single Source Shortest Paths (SSSP)

Dijkstra's Algorithm

Initialization

$p(v) \leftarrow \text{null}, d(v) \leftarrow \infty$ for all $v \neq s$

$p(s) \leftarrow \text{null}, d(s) \leftarrow 0$

set $S \leftarrow \emptyset$

insert all vertices v into priority queue Q with key $d(v)$

Main Loop

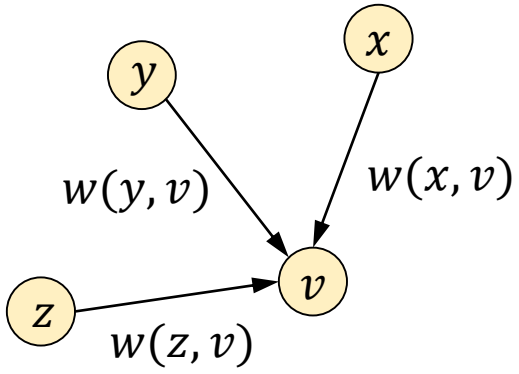
while Q is not empty {
    $u \leftarrow \text{Q.delMin}()$
    $S \leftarrow S \cup \{u\}$
    for all edges (u, v) {
        $\text{relax}(u, v)$
    }
}

priority queue Q	running time
array	$O(n^2)$
binary heap	$O(m \log n)$
Fibonacci heap	$O(m + n \log n)$

Single Source Shortest Paths (SSSP) in Map-Reduce

- Not easy to parallelize Dijkstra's algorithm
- Use an iterative approach instead
  - The distance $d(v)$ from s to v is updated by the distances of all u with $(u, v) \in E$.

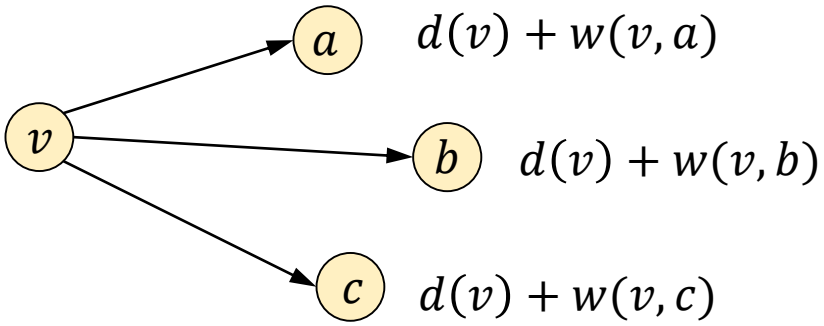
$$d(v) \leftarrow \min\{d(u) + w(u, v) | \, (u, v) \in E\}$$



  - Need to communicate both distances and adjacency lists.

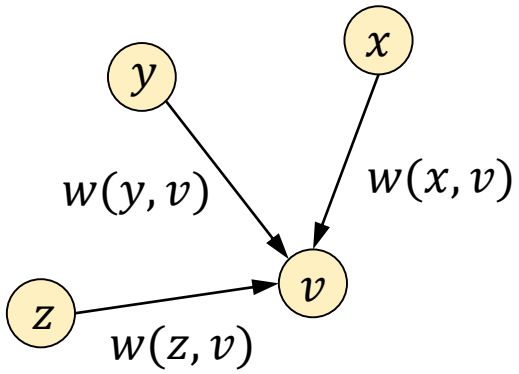
Single Source Shortest Paths (SSSP) in Map-Reduce

Mapper: emits distances and graph structure



Reducer: updates distances and emits graph structure

$$d(v) \leftarrow \min\{d(u) + w(u, v) | \ (u, v) \in E\}$$



Single Source Shortest Paths (SSSP) in Map-Reduce

- Not easy to parallelize Dijkstra's algorithm
- Use an iterative approach instead
  - The distance $d(v)$ from s to v is updated by the distances of all u with $(u, v) \in E$.
  - Need to communicate both distances and adjacency lists.
  - Repeat round until all distances are fixed.
  - Number of rounds = $n - 1$ in the worst case.
  - If all weights are equal then we compute the Breadth-First Search (BFS) tree. Number of rounds = graph diameter.

BFS in Map-Reduce

```
1: class Mapper
2:     method Map(nid n, node N)
3:         d ← N.Distance
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, d + 1)                  ▷ Emit distances to reachable nodes

1: class Reducer
2:     method Reduce(nid m, [d1, d2, ...])
3:         d_min ← ∞
4:         M ← ∅
5:         for all d ∈ counts [d1, d2, ...] do
6:             if IsNode(d) then
7:                 M ← d                           ▷ Recover graph structure
8:             else if d < d_min then              ▷ Look for shorter distance
9:                 d_min ← d
10:        M.Distance ← d_min                      ▷ Update shortest distance
11:        Emit(nid m, node M)
```

Single Source Shortest Paths (SSSP) in Map-Reduce

Remarks on Map-Reduce SSSP algorithm

- Essentially a brute-force algorithm.
- Performs many unnecessary computations.
- No global data structure.

PageRank in Map-Reduce

Recall the formula for the PageRank $R(u)$ of a webpage u

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v} + (1 - c)E_u$$

B_u = set of pages that point to u

F_u = set of pages that u points to

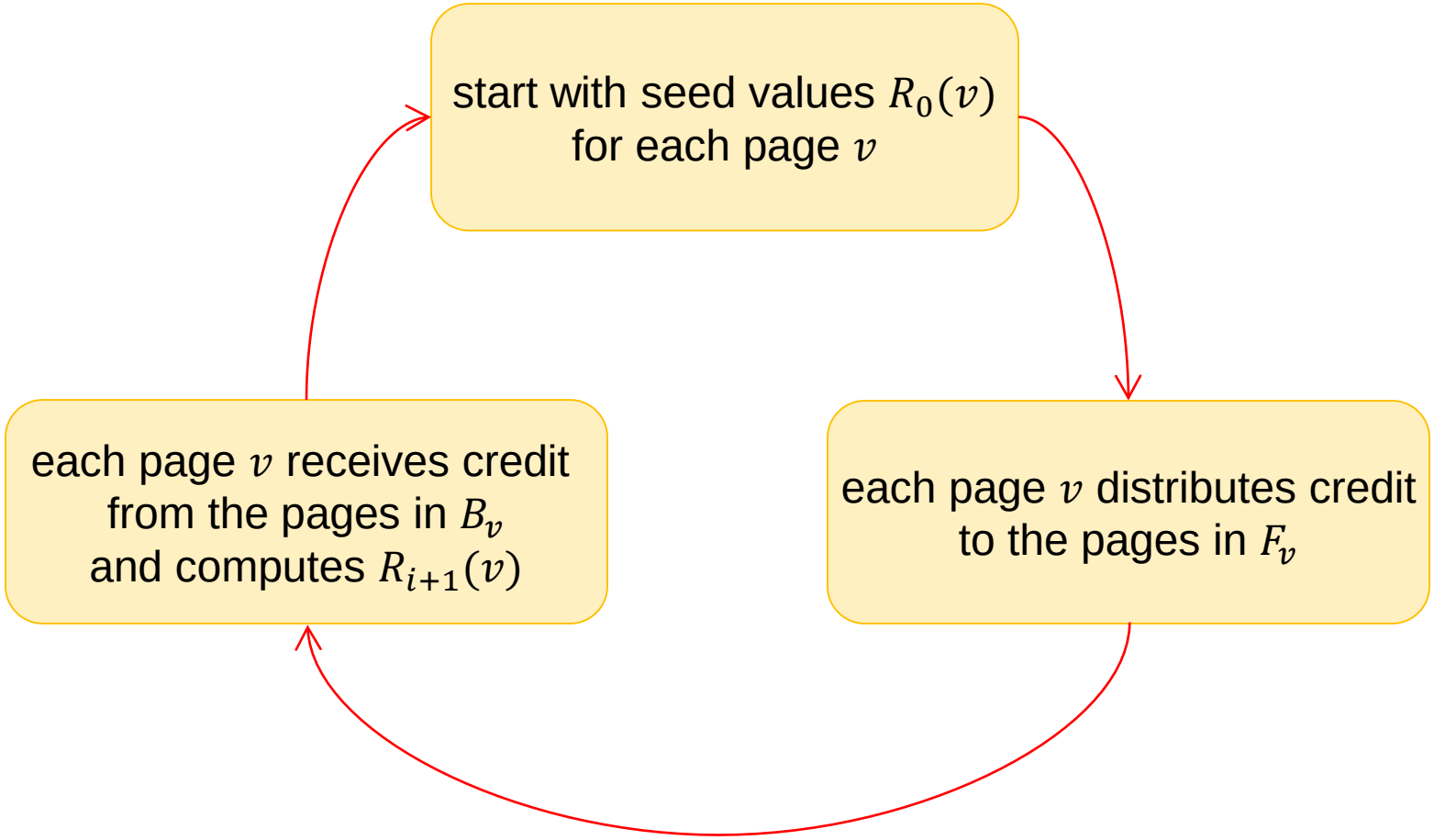
$|F_u| = N_u$ = number of links from u

E_u = probabilities over web pages

E_u and c are user designed parameters

PageRank in Map-Reduce

Iterative computation



PageRank in Map-Reduce

```
1: class Mapper
2:     method Map(nid n, node N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, p)                      ▷ Pass PageRank mass to neighbors

1: class Reducer
2:     method Reduce(nid m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ counts [p1, p2, ...] do
5:             if IsNode(p) then
6:                 M ← p                           ▷ Recover graph structure
7:             else
8:                 s ← s + p                       ▷ Sums incoming PageRank contributions
9:         M.PageRank ← s
10:        Emit(nid m, node M)
```

Algorithms and Complexity in MapReduce (and related models)

Sorting, Searching, and Simulation in the MapReduce Framework
M. T. Goodrich, N. Sitchinava, and Q. Zhang
ISAAC 2011

Fast Greedy Algorithms in MapReduce and Streaming
R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani
SPAA 2013

On the Computational Complexity of MapReduce
B. Fish, J. Kun, A. D. Lelkes, L. Reyzin, and G. Turan
DISC 2015

BSP model

L. G. Valiant, A Bridging Model for Parallel Computation,
Communications of the ACM, 1990

Computational model of parallel computation

BSP is a parallel programming model based on Synchronizer Automata.
The model consists of:

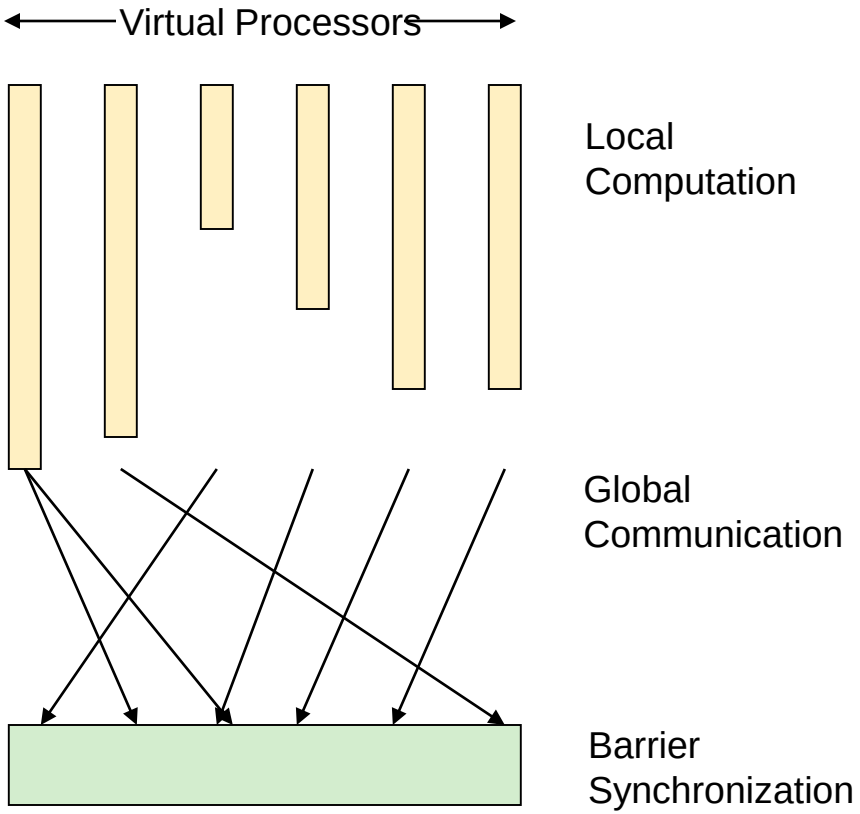
- Set of processor-memory pairs.
- Communications network that delivers messages in a **point-to-point** manner.
- Mechanism for the efficient **barrier synchronization** for all or a subset of the processes.
- No special combining, replicating, or broadcasting facilities.

BSP model

- Vertical Structure

  Supersteps:
  - Local computation
  - Process Communication
  - Barrier Synchronization

- Horizontal Structure
  - Concurrency among a fixed number of virtual processors.
  - Processes do not have a particular order.
  - Locality plays no role in the placement of processes on processors.



Implementation: BSPlib

MapReduce simulation of a BSP program

Simulation on MapReduce:

1. Create a tuple for each memory cell and processor.
2. Map each message to the destination processor label.
3. Reduce by performing one step of a processor, outputting the messages for next round.

Theorem [Goodrich et al.]: Given a BSP algorithm A that runs in T supersteps with a total memory size N using $P \leq N$ processors, we can simulate A using $O(T)$ rounds and message complexity $O(TN)$ in the memory-bound MapReduce framework with reducer memory size bounded by N/P.

MapReduce simulation of a BSP program

Simulation on MapReduce:

1. Create a tuple for each memory cell and processor.
2. Map each message to the destination processor label.
3. Reduce by performing one step of a processor, outputting the messages for next round.

Theorem [Goodrich et al.]: Given a BSP algorithm A that runs in T supersteps with a total memory size N using $P \leq N$ processors, we can simulate A using $O(T)$ rounds and message complexity $O(TN)$ in the memory-bound MapReduce framework with reducer memory size bounded by N/P.

A corollary of the above:
Given the optimal BSP algorithm of [Goodrich, 99], we can sort N values in the MapReduce framework in $O(k)$ rounds and $O(kN)$ message complexity.

Algorithms and Complexity in MapReduce (and related models)

Theorem [Fish et al.] : Any problem requiring sublogarithmic space, $o(\log n)$, can be solved in MapReduce in two rounds.

The proof is **constructive**: Given a problem that classically takes less than logarithmic space, there is an **automatic** algorithm to implement it in MapReduce