Anthony Ventresque
anthony.ventresque@ucd.ie

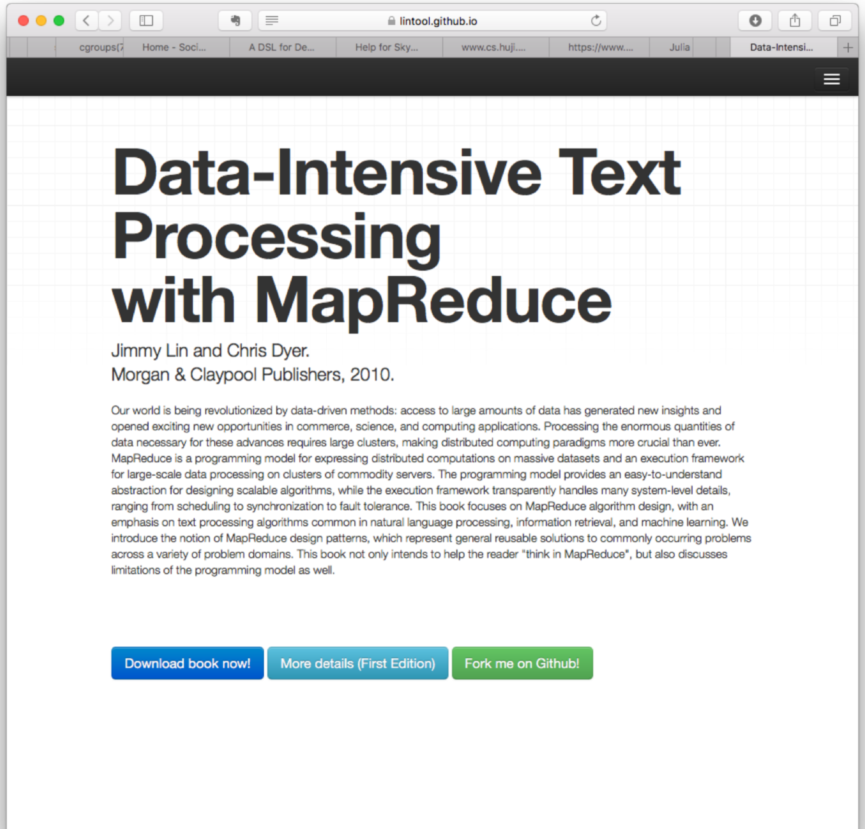
Big Data Programming

COMP47470

Graphs

School of Computer Science, UCD   Scoil na Ríomheolaíochta, UCD

Recommended Reading



https://lintool.github.io/MapReduceAlgorithms/

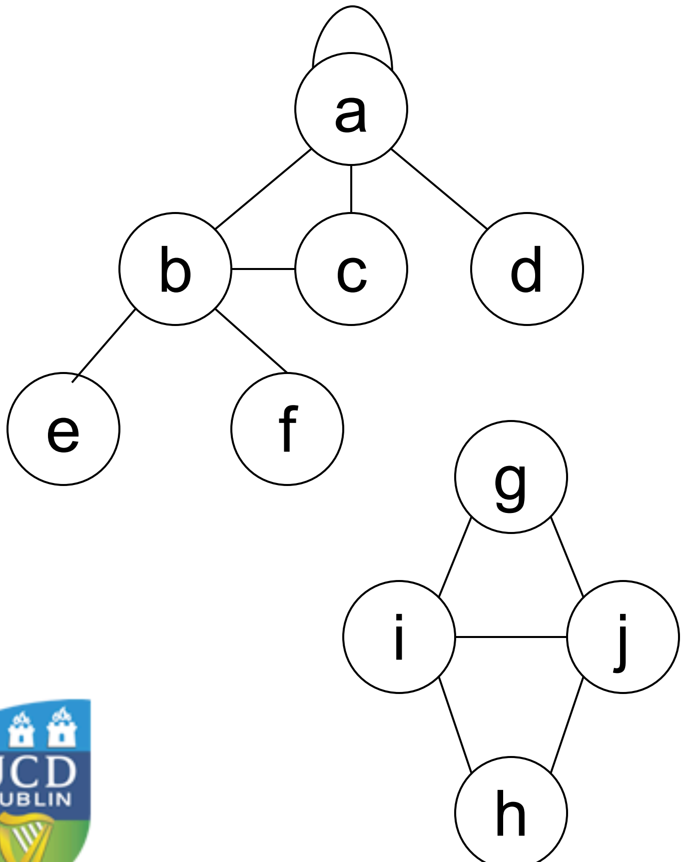
Graphs

- Ubiquitous in modern society
  - Hyperlink structure of the Web
  - Social networks
    - Email flow
    - Friend patterns
  - Transportation networks
- Nodes and links can be annotated with metadata
  - Social network nodes: age, gender, interests
  - Social network edges: relationship type and importance

Real-world Problems to Solve

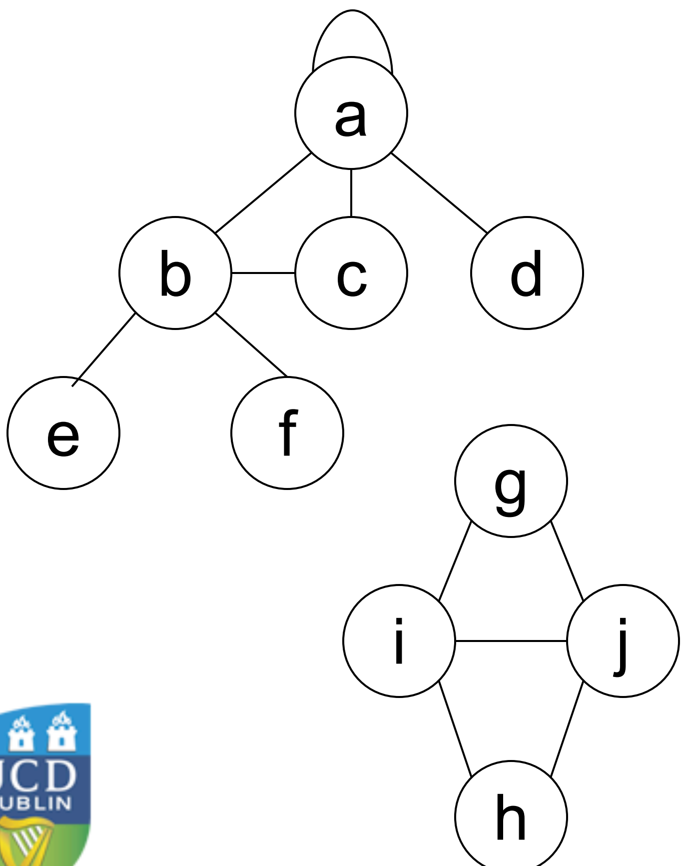
- A common feature: millions or billions of nodes and millions or billions of edges
- Real-world graphs are often sparse, the number of actual edges is far smaller than the number of possible edges

Adjacency List



a	a, b, c, d
b	a, c, e, f
c	a, b
d	a
e	b
f	b
g	i, j
h	i, j
i	g, h, j
j	g, h, i

Adjacency Matrix



	a	b	c	d	e	f	g	h	i	j
a	1	1	1	1						
b	1		1		1	1				
c	1	1								
d	1									
e		1								
f		1								
g									1	1
h									1	1
i							1	1		1
j							1	1	1	

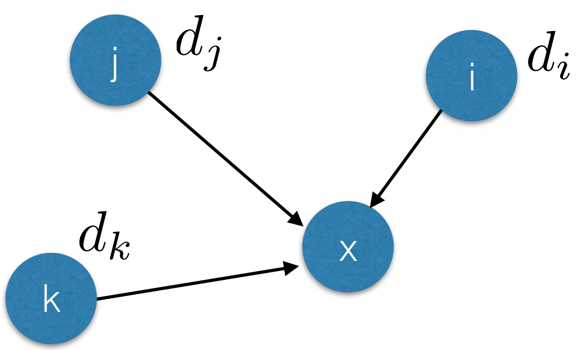
Comparison

- Adjacency Matrix: mathematically easy representation but waste of space
- Adjacency List: A much more compressed representation (for sparse graphs) but some graph operations are more difficult compared to the adj. matrix

- Counting inlinks:
  - Matrix: scan the column and count
  - List: difficult, worst case all data needs to be scanned
- Counting outlinks:
  - Matrix: scan the rows and count
  - List: outlinks are natural

Shortest path in the MapReduce World

- Task: find the shortest path from a source node to all other nodes in the graph. Edges have unit weight.
- Intuition:
  - Distance of nodes *N* directly connected to the source is 1
  - Distance of nodes directly connected to nodes in N is 2
  - Multiple path to *x*: the shortest path must go through one of the nodes with an outlink to *x*; use the minimum



$$d_x = min(d_i + 1, d_j + 1, d_k + 1)$$

Shortest path in the MapReduce World

```
class Mapper
    method Map(nid n, node N)
        d ← N.Distance
        Emit(nid n, N)
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, d + 1)

class Reducer
    method Reduce(nid m, [d1, d2, ...])
        dmin ← ∞
        M ← ∅
        for all d ∈ counts [d1, d2, ...] do
            if IsNode(d) then
                M ← d
            else if d < dmin then
                dmin ← d
        M.Distance ← dmin
        Emit(nid m, node M)
```

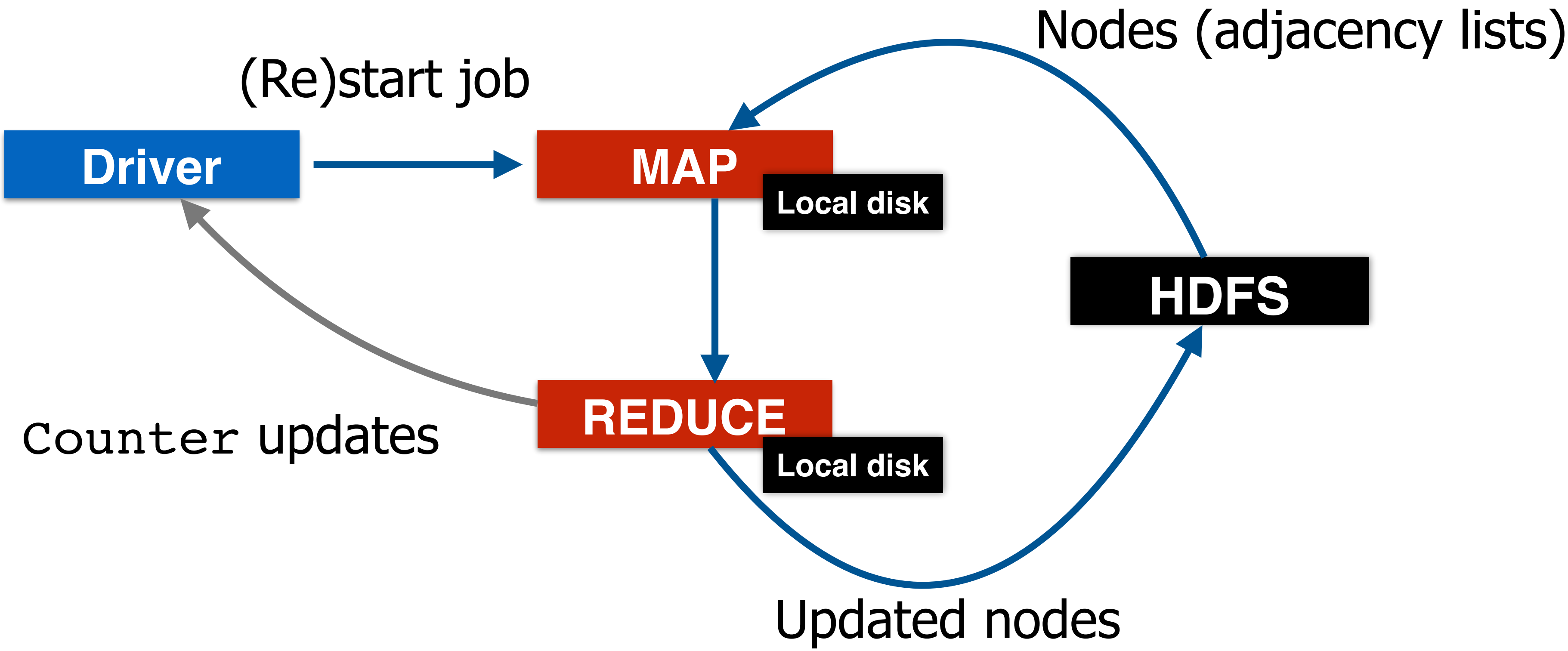
Shortest path in the MapReduce World

- Each iteration of the algorithm is one Hadoop job
  - A map phase to compute the distances
  - A reduce phase to find the current minimum distance
- Iterations:
  1. All nodes connected to the source are discovered
  2. All nodes connected to those discovered in 1. are found
  3. 3. ...
- Between iterations (jobs) the graph structure needs to be passed along; reducer output is input for the next iteration

Shortest path in the MapReduce World

- How many iterations are necessary to compute the shortest path to all nodes?
  - Diameter of the graph (greatest distance between a pair of nodes)
  - Diameter is usually small ("six degrees of separation")
- In practice: iterate until all node distances are less than +infinity
  - Assumption: connected graph
- Termination condition checked "outside" of MapReduce job
  - Use Counter to count number of nodes with infinite distance

Shortest path in the MapReduce World



Shortest path in the MapReduce World

- What if the edges have weights?

- Two changes required wrt. the parallel BFS
  - Update rule, instead of *d+1* use *d+w*
  - Termination criterion: no more distance changes (via Counter)
- Num. iterations in the worst case: *#nodes-1*

Single-source Shortest Path

- Dijkstra
  - Single processor (global data structure)
  - Efficient (no recompilation of finalised states)

- Parallel BFS
  - Brute Force approach
  - A lot of unnecessary computations (distances to all nodes recomputed at each iteration)
  - no global data structure

Prototypical approach to graph algorithms in MapReduce/Hadoop

- Node datastructure which contains
  - Adjacency list
  - Additional node [and possibly edge] information (type, features, distances, weights, etc.)
- Job maps over the node data structures
  - Computation involves a node's internal state and local graph structure
  - Result of map phase emitted as values, keyed with node ids of the neighbours; reducer aggregates a node's results
- Graph itself is passed from Mapper to Reducer
- Algorithms are iterative, requiring several Hadoop jobs controlled by the driver code

Outline

- The Web Graph
- PageRank
- Issues and Solutions

THE WEB GRAPH

The Web's Graph Structure

Graph structure in the Web

Andrei Broder [a], Ravi Kumar [b,*], Farzin Maghoul [a], Prabhakar Raghavan [b],
Sridhar Rajagopalan [b], Raymie Stata [c], Andrew Tomkins [b], Janet Wiener [c]

[a] AltaVista Company, San Mateo, CA, USA
[b] IBM Almaden Research Center, San Jose, CA, USA
[c] Compaq Systems Research Center, Palo Alto, CA, USA

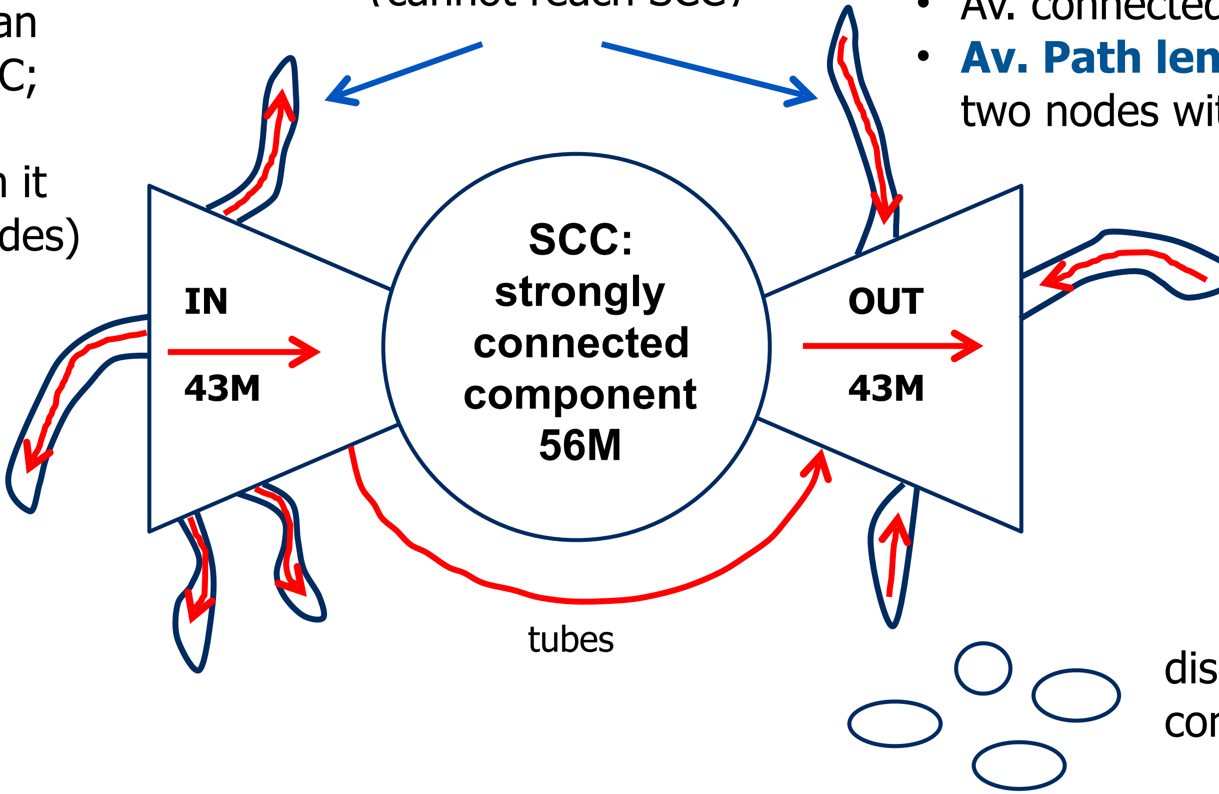
- Insights important for:
  - Crawling strategies
  - Analyzing the behaviour of algorithms that rely on link information (such as PageRank)
  - Predicting the evolution of Web structures
  - etc.
- Data: Altavista crawl from 1999 with 200 million pages and 1.5 billion links

The Web as a "Bow Tie"

- ~200M nodes in total
- **>90% in a single WCC**
- **Av. connected distance SCC: 28**
- Av. connected distance graph: >500
- **Av. Path length: 16** between any two nodes with existing path

nodes that can reach the SCC; cannot be reached from it (e.g. new nodes)

tendrils (44M)
(cannot reach SCC)



nodes that can reach be reached from the SCC but do not link back (e.g. corporate nodes)

disconnected components (17M)

PageRank

- A topic independent approach to page importance
  - Computed once per crawl
- Every document of the corpus is assigned an importance score
  - In search: re-rank (or filter) results with a low PageRank score
- Simple idea: number of in-link indicates importance
  - Page p1 has 10 in-links and one of those is from yahoo.com,
  - page p2 has 50 in-links from obscure pages
- PageRank takes the importance of the page where the link originates into account

"To test the utility of PageRank for search, we built a web search engine called Google."

PageRank

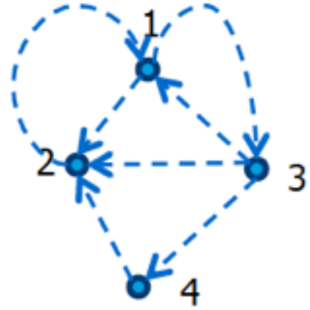
- Idea: if page px links to page py, then the creator of px implicitly transfers some importance to page py
  - yahoo.com is an important page, many pages point to it
  - Pages linked to from yahoo.com are also likely to be important
- Pages distribute "importance" through outlinks
- Simple PageRank (iteratively)

out-degree of node u

$$PageRank_{i+1}(v) = \sum_{u \rightarrow v} \frac{PageRank_i(u)}{N_u}$$

all nodes linking to v

PageRank



$$W = \begin{pmatrix} \boxed{\begin{matrix} 0 \\ 1/2 \\ 1/2 \\ 0 \end{matrix}} & \begin{matrix} 1 \\ 0 \\ 0 \\ 0 \end{matrix} & \begin{matrix} 1/3 \\ 1/3 \\ 0 \\ 1/3 \end{matrix} & \begin{matrix} 0 \\ 1 \\ 0 \\ 0 \end{matrix} \end{pmatrix}$$

outlinks of node 1

$$w_{ij} : \frac{link\ j \rightarrow i}{outlinks_j}$$

Simplified formula

initialize PageRank vector $\vec{R}$

$$\vec{R} = (R(1), \ldots, R(4)) = (0.25, 0.25, 0.25, 0.25)$$

$$PageRank_i = W \times PageRank_{i-1}$$

$$W^1 \times \vec{R}' = \begin{pmatrix} 0.33 \\ 0.46 \\ 0.13 \\ 0.08 \end{pmatrix}$$

$$W^2 \times \vec{R}' = \begin{pmatrix} 0.50 \\ 0.29 \\ 0.17 \\ 0.04 \end{pmatrix}$$

$$W^3 \times \vec{R}' = \begin{pmatrix} 0.35 \\ 0.35 \\ 0.25 \\ 0.06 \end{pmatrix}$$

PageRank vector converges eventually

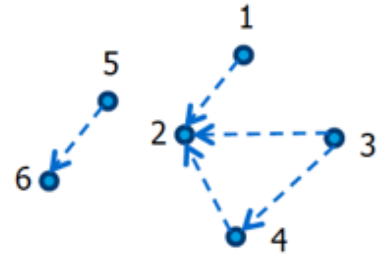
$$W^{16} \times \vec{R}' = \begin{pmatrix} 0.40 \\ 0.33 \\ 0.20 \\ 0.07 \end{pmatrix}$$

$$W^{17} \times \vec{R}' = \begin{pmatrix} 0.40 \\ 0.34 \\ 0.20 \\ 0.07 \end{pmatrix}$$

Random surfer model:

- Probability that a random surfer starts at a random page and ends at page $\mathbf{p_x}$
- A random surfer at a page with 3 outlinks randomly picks one (1/3 prob.)

PageRank



$$W = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \end{pmatrix}$$

Reality

disconnected components

nodes without outgoing edges lead to problems (**rank sink**)

initialize PageRank vector $\vec{R}$

$$\vec{R} = (R(1),\ldots,R(4)) = (0.25, 0.25, 0.25, 0.25)$$

$$W^1 \times \vec{R}' = \begin{pmatrix} 0.00 \\ 0.63 \\ 0.00 \\ 0.13 \end{pmatrix}$$

$$W^2 \times \vec{R}' = \begin{pmatrix} 0.00 \\ 0.13 \\ 0.00 \\ 0.00 \end{pmatrix}$$

$$W^3 \times \vec{R}' = \begin{pmatrix} 0.00 \\ 0.00 \\ 0.00 \\ 0.00 \end{pmatrix}$$

Include a decay ("damping") factor

$$PageRank_{i+1}(v) = \alpha \left(\frac{1}{|G|} \right) + (1 - \alpha) \sum_{u \to v} \frac{PageRank_i(u)}{N_u}$$

probability that the random surfer "**teleports**" and not uses the outlinks

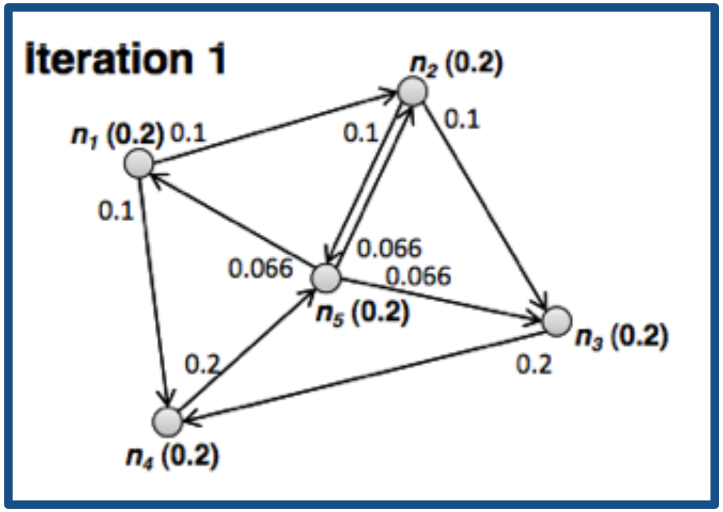
PageRank in MapReduce

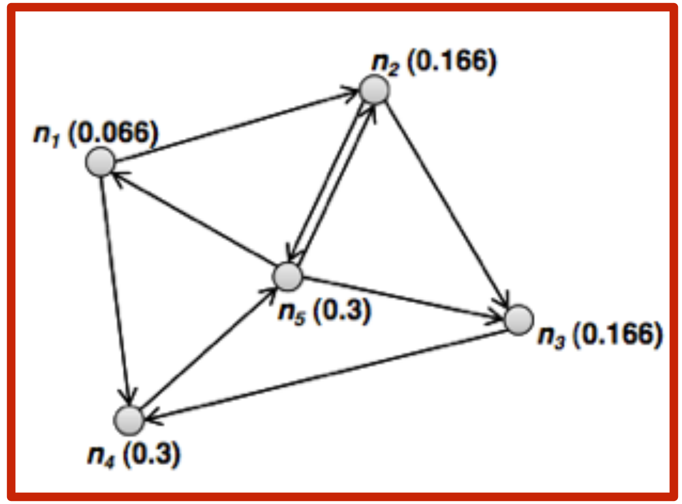
- At each iteration:
  - **[MAPPER]** a node passes its PageRank "contributions" to the nodes it is connected to
  - **[REDUCER]** each node sums up all PageRank contributions that have been passed to it and updates its PageRank score

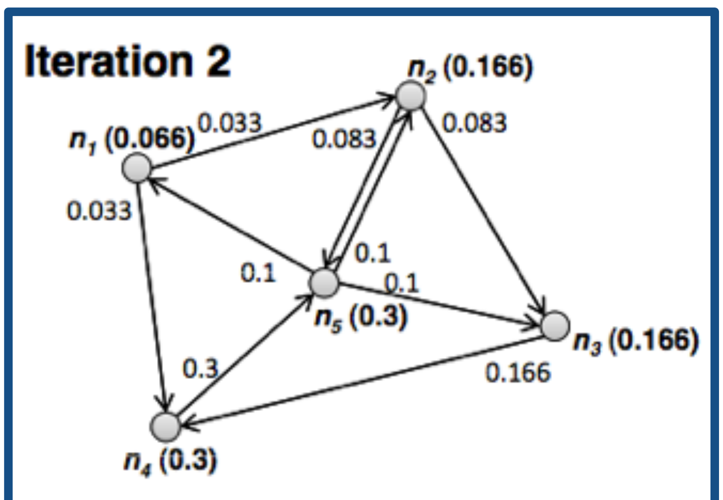
PageRank in MapReduce

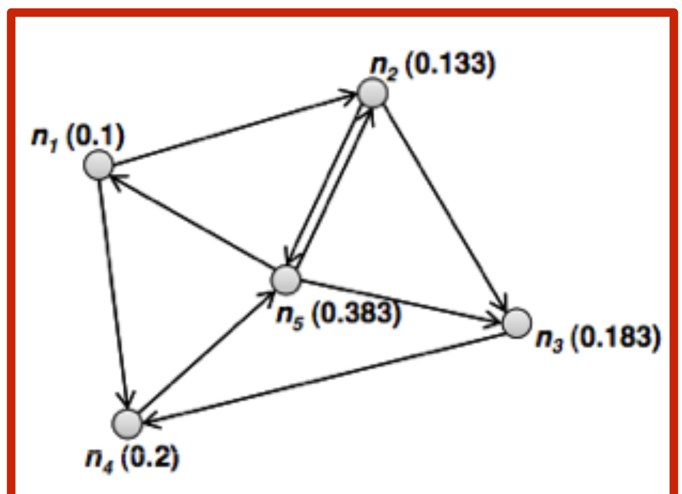
An informal sketch

$$\alpha = 0, \sum_{i=1}^{5} n_i = 1$$









PageRank in MapReduce

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

class Reducer
    method Reduce(nid m, [p1, p2, ...])
        M ← ∅
        for all p ∈ counts [p1, p2, ...] do
            if IsNode(p) then
                M ← p                                   ▷ Recover graph structure
            else
                s ← s + p                               ▷ Sum incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```

PageRank in MapReduce

- Dangling nodes: nodes without outgoing edges
  - Simplified PR cannot conserve total PageRank mass (black holes for PR scores)
  - Solution: “lost” PR scores are redistributed evenly across all nodes in the graph
  - Use Counters to keep track of lost mass
  - Reserve a special key for PR mass from dangling nodes
- Redistribution of lost mass and jump factor after each PR iteration in another job (MAP phase only job)

PageRank in MapReduce

- (Possible) stopping criteria
  - PageRank is iterated until convergence (scores at nodes no longer change)
  - PageRank is run for a fixed number of iterations
  - PageRank is run until the ranking of the nodes according to their PR score no longer changes
  - Original PageRank paper: 52 iterations until convergence on a graph with more than 300M edges

ISSUES AND SOLUTIONS

Efficient Large-scale Graph Processing is Challenging

- Poor locality of memory access
- Little work per node (vertex)
- Changing degree of parallelism over the course of execution
- Distribution over many commodity machines due to poor locality is error-prone (failure likely)
- Needed: "*scalable general-purpose system for implementing arbitrary graph algorithms [in batch mode] over arbitrary graph representations in a large-scale distributed environment*"

Existing graph processing options (until 2010)

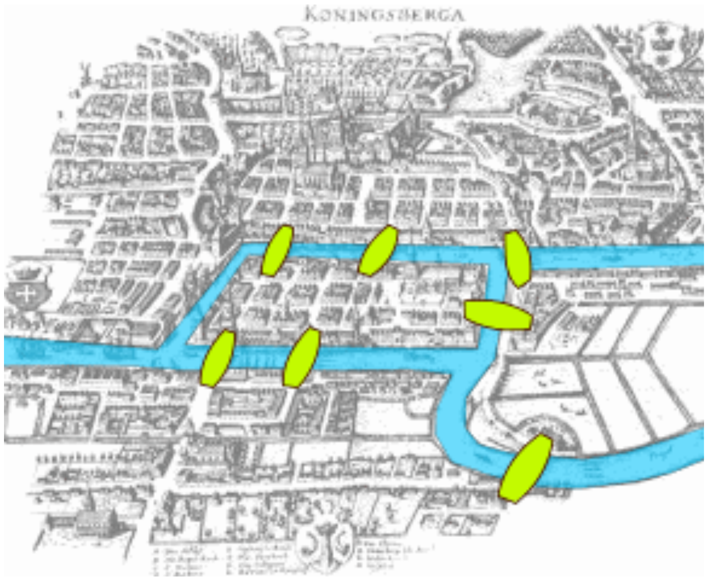
- Custom distributed infrastructure
  - Problem: each algorithm requires new implementation effort
- Relying on the MapReduce framework
  - Problem: performance and usability issues
  - Remember: the whole graph is read/written in every job
- Single-processor graph algorithm library (e.g. LEDA)
  - Problem: does not scale
- Existing parallel graph systems
  - Problem: do not address fault tolerance & related issues appearing in large distributed setups

Enter Pregel (2010)

Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn,
Naty Leiser, and Grzegorz Czajkowski
Google, Inc.
{malewicz,austern,ajcbik,dehnert,ilan,naty,gczaj}@google.com

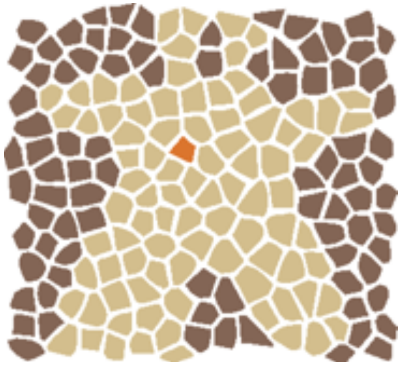
- "We built a scalable and fault-tolerant platform with an API that is sufficiently flexible to express arbitrary graph algorithms"
- Pregel river runs through Königsberg (Euler's seven bridges problem)



Graph processing in Hadoop

- Disadvantage: iterative algorithms are slow
  - Lots of reading/writing to and from disk
- Advantage: no additional libraries needed

- Enter Giraph: an open-source implementation of yet another Google framework (Pregel)
  - Specifically created for iterative graph computations



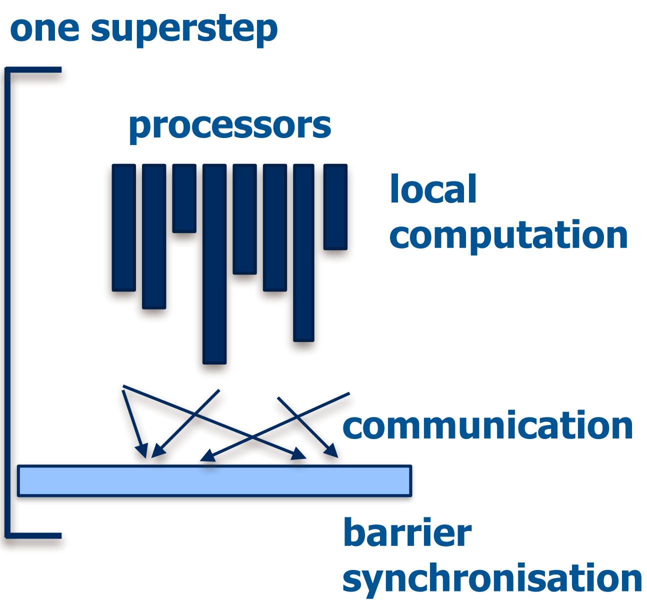
Graph processing in Hadoop

- “Many distributed graph computing systems have been proposed to conduct all kinds of data processing and data analytics in massive graphs, including Pregel, Giraph, GraphLab, PowerGraph, GraphX, Mizan, GPS, Giraph++, Pregelix, Pregel+, and Blogel.”

BULK SYNCHRONOUS PARALLEL -- BSP

Bulk Synchronous Parallel

- General model for the design of parallel algorithms
- Developed by Leslie Valiant 1980s/90s
- BSP computer: processors with fast local memory are connected by a communication network
- BSP computation = series of supersteps



- No message passing in MR
- Avoids MR's costly disk and network operations

Bulk Synchronous Parallel

- Supersteps consist of three phases
  1. Local computation: every processor performs computations using data stored in local memory - independent of what happens at other processors; a processor can contain several processes (threads)
  2. Communication: exchange of data between processes (put and get); one-sided communication
  3. Barrier synchronisation: all processes wait until everyone has finished the communication step

- Local computation and communication phases are not strictly ordered in time

Bulk Synchronous Parallel

- BSP & graphs: “**Think like a vertex!**”
- In BSP, algorithms are implemented from the viewpoint of a **single vertex** in the input graph performing a **single iteration** of the computation.