# Lecture 11: Graph algorithms

Claudia Hauff (Web Information Systems)
ti2736b-ewi@tudelft.nl

# Course content

- Introduction
- Data streams 1 & 2
- The MapReduce paradigm
- Looking behind the scenes of MapReduce: HDFS & Scheduling
- Algorithm design for MapReduce
- A high-level language for MapReduce: Pig Latin 1 & 2
- MapReduce is not a database, but HBase nearly is
- **Lets iterate a bit: Graph algorithms** & Giraph
- How does all of this work together? ZooKeeper/Yarn

# Learning objectives

- Give examples of real-world problems that can be solved with graph algorithms
- **Explain** the major differences between BFS on a single machine (Dijkstra) and in a MapReduce framework
- **Explain** the main ideas behind PageRank
- **Implement** iterative graph algorithms in Hadoop

# Graphs

# Graphs

- Ubiquitous in modern society
  - Hyperlink structure of the Web
  - Social networks
    - Email flow
    - Friend patterns
  - Transportation networks

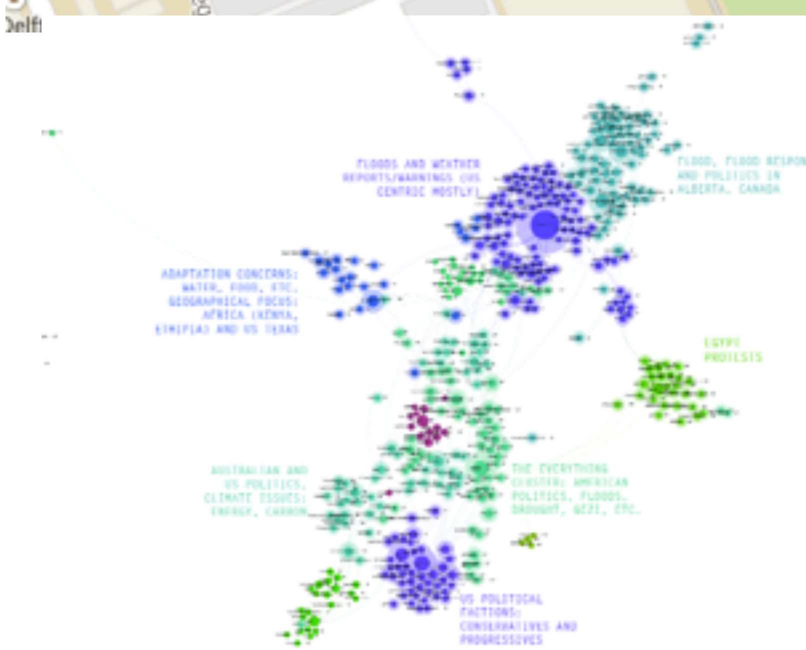
- Nodes and links can be annotated with **metadata**
  - Social network nodes: age, gender, interests
  - Social network edges: relationship type (friend, spouse, foe, etc.), relationship importance (weights)

# Real-world problems to solve

- Graph search
  - Friend recommend. in social networks
  - Expert finding in social networks

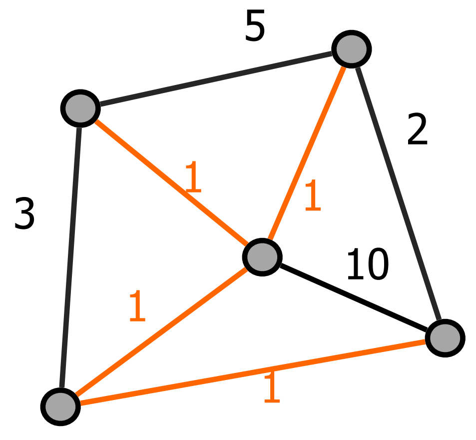

- Path planning
  - Route of network packets
  - Route of delivery trucks


- Graph clustering
  - Subcommunities in large graphs

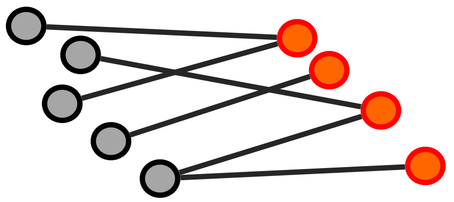


# Real-world problems to solve

- **Minimum spanning tree**: a tree that contains all vertices of a graph and the cheapest edges
  - Laying optical fiber to span a number of destinations at the lowest possible cost




- **Bipartite graph matching**: two disjoint vertex sets
  - Job seekers looking for employment
  - Singles looking for dates



# Real-world problems to solve

- Identification of special nodes
  - Special based on various metrics (in-degree, average distance to other nodes, relationship to the cluster structure, …)

- Maximum flow
  - Compute traffic that can be sent from source to sink given various flow capacity constraints

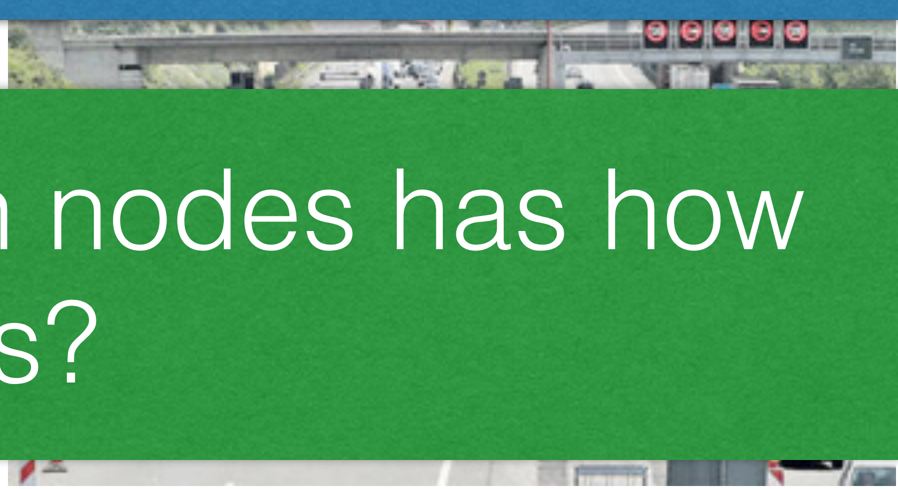
# Real-world problems to solve

- Identification of special nodes

A common feature: millions or billions of nodes & 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.

flow capacity constraints

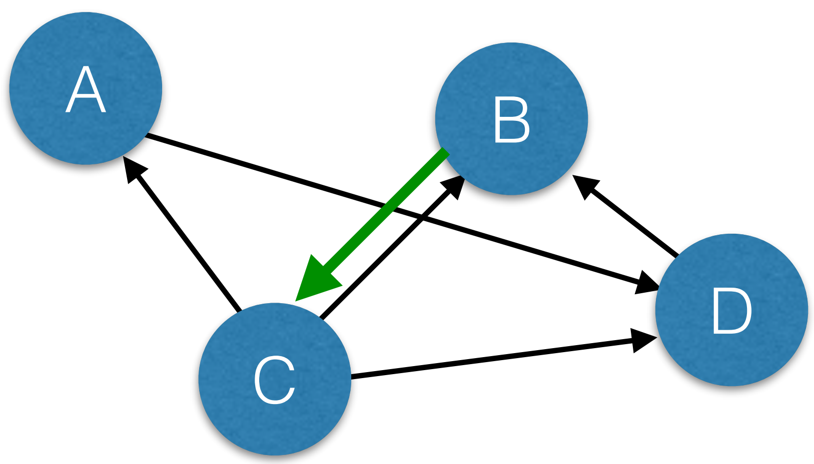


Question: a friendship graph with n nodes has how many possible edges?

# A bit of graph theory

# Connected components

- **Strongly** connected component (SCC): directed graph with a path from each node to every other node

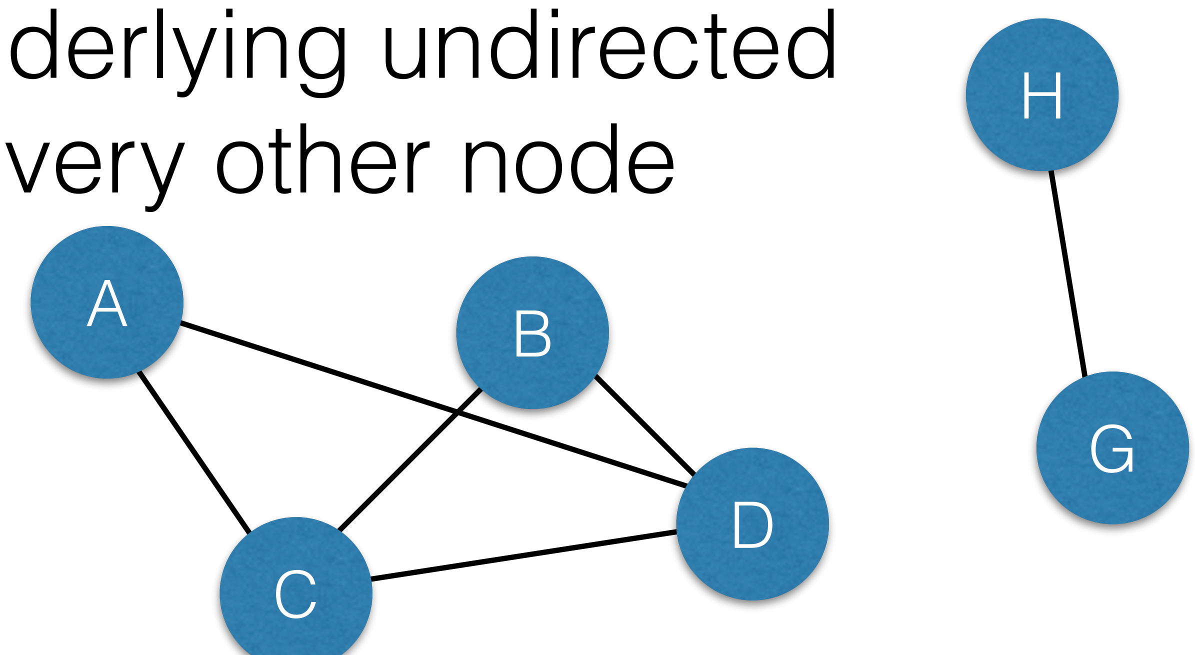



A cannot reach C
B cannot reach A
.....

strongly connected

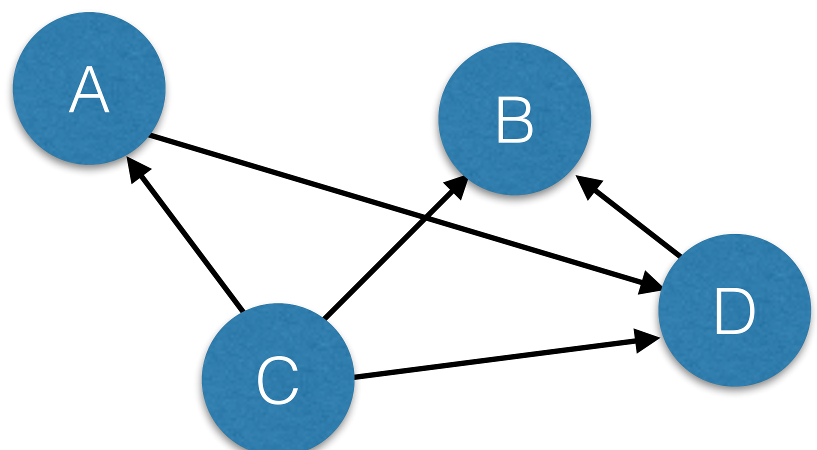
- **Weakly** connected component (WCC): directed graph with a path in the underlying undirected graph from each node to every other node

2 weakly connected components



# Connected components

- **Strongly** connected component (SCC): directed graph with a path from each node to every other node




$$G = (V, E)$$

graph

$$V = \{A, B, C, D\}$$

nodes

**directed** edges

$$E = \{(A, D), (B, C), (C, A), (C, B), (C, D), (D, B)\}$$

$$d(A, B) = 2, d(C, B) = 1, d(A, C) = 3$$

**shortest** distance between 2 nodes

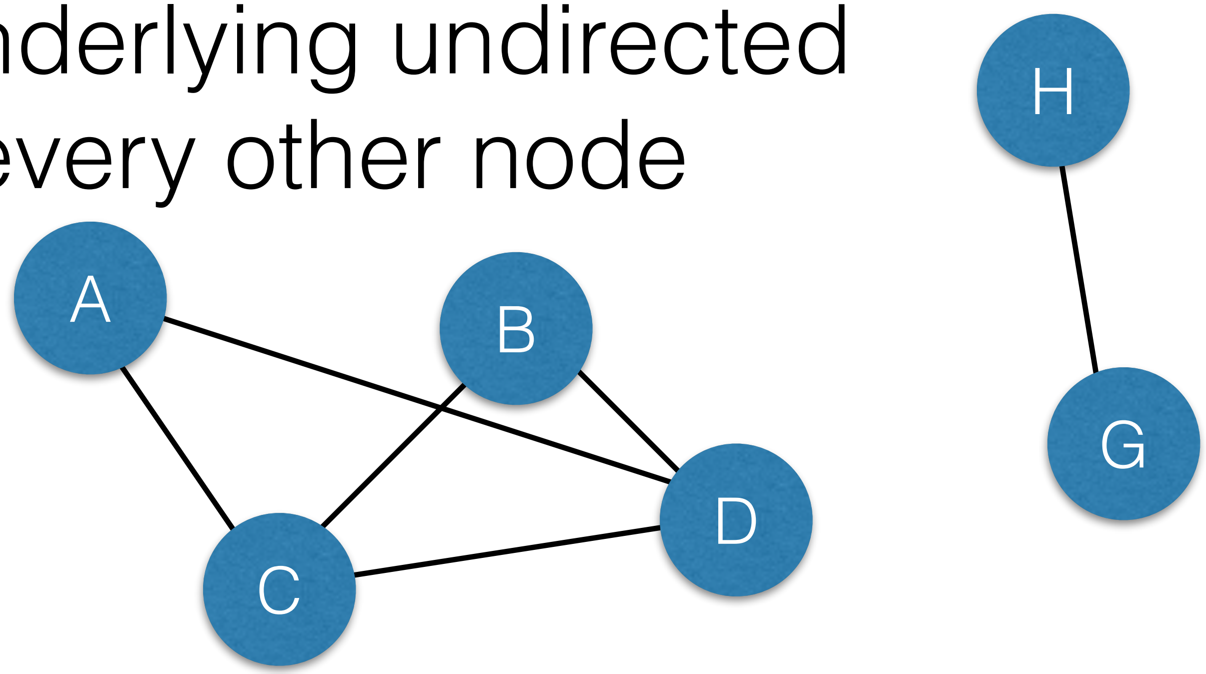
# Connected components

$$G = (V, E)$$
$$V = \{A, B, C, D, G, H\}$$
$$E = \{\{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}, \{G, H\}\}$$
$$d(A, B) = 2, d(C, B) = 1, d(A, C) = 1, d(A, G) = \infty$$

**undirected** edges

**infinite** distance

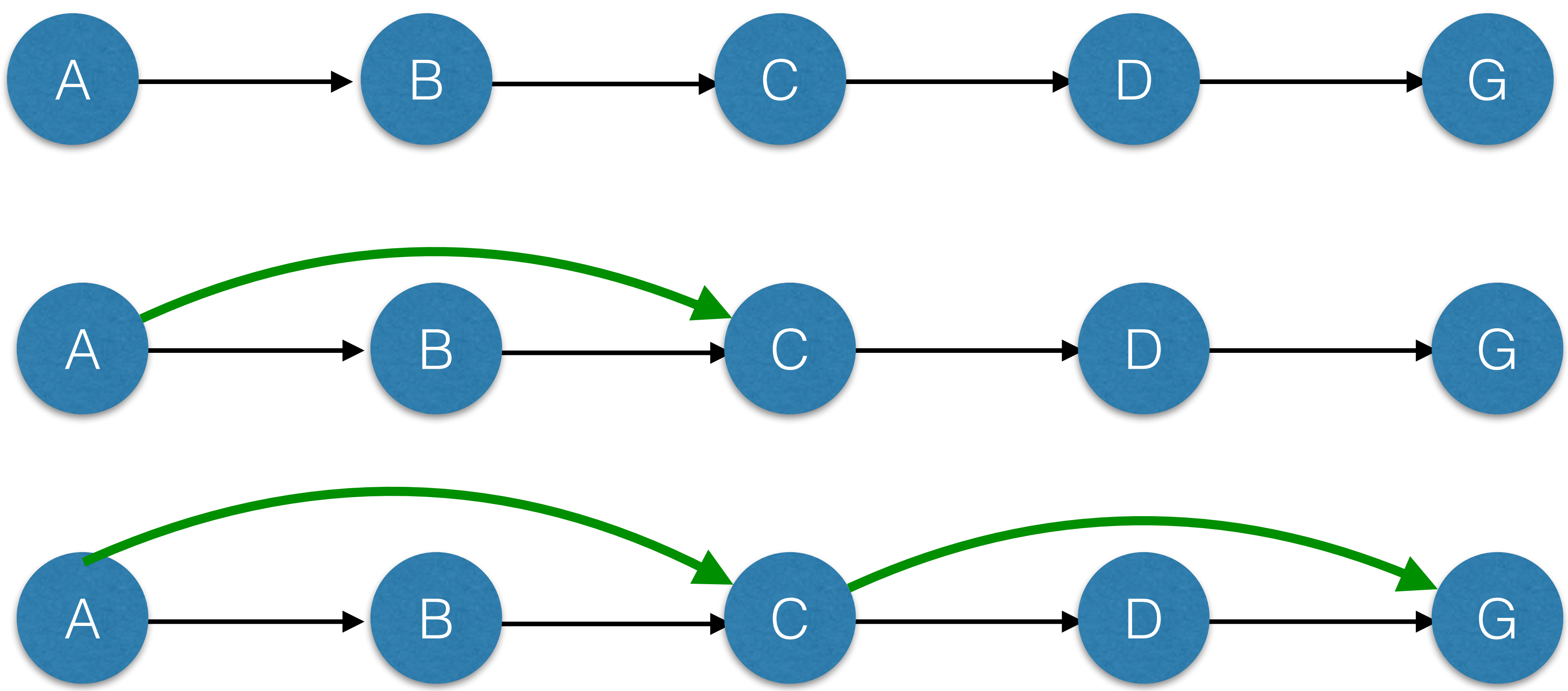
- **Weakly** connected component (WCC): directed graph with a path in the underlying undirected graph from each node to every other node



# Graph diameter

Definition: **longest shortest path** in the graph

$$max_{x,y \in V} d(x, y)$$



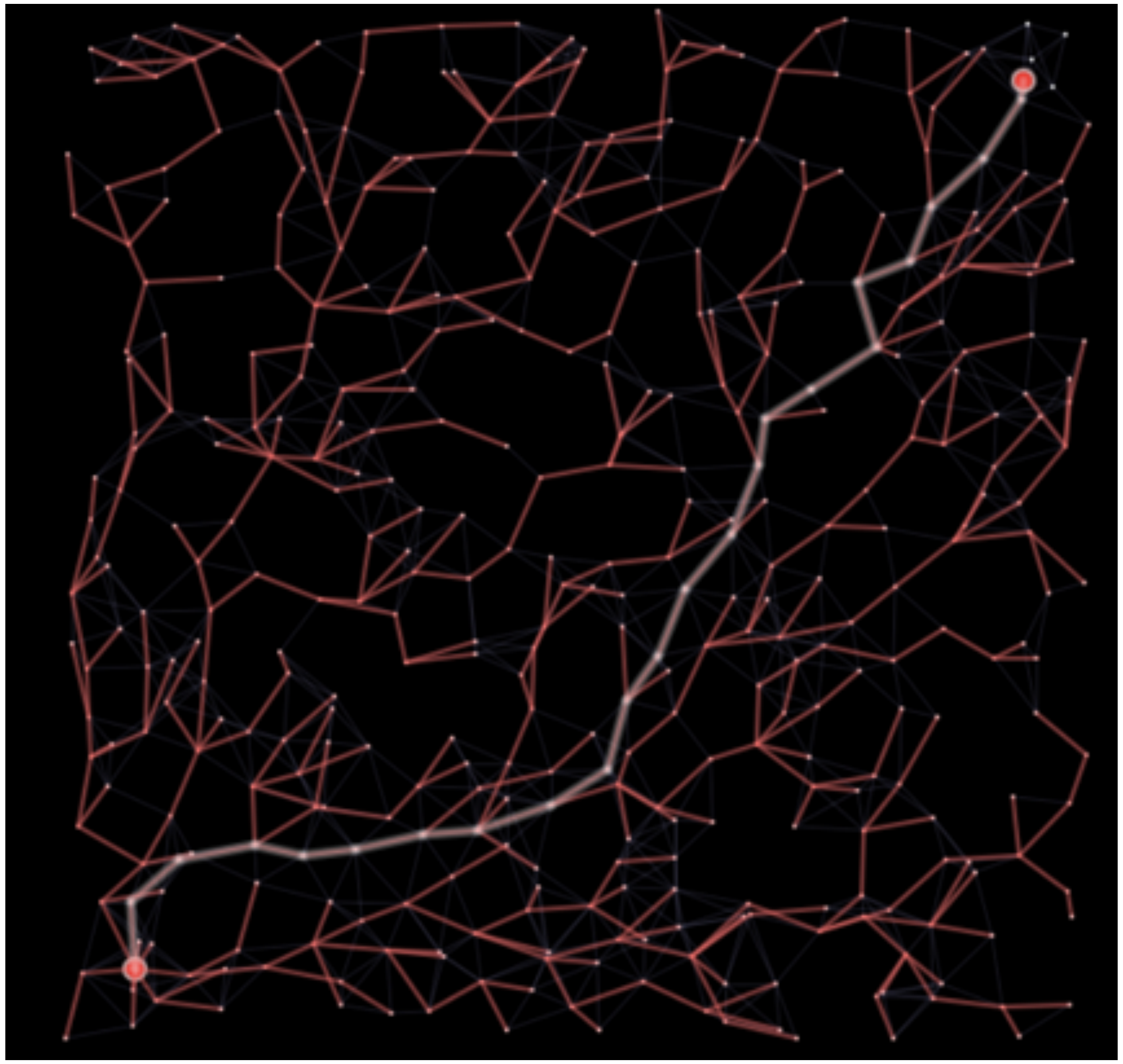

diameter: 4

diameter: 3

diameter: 2

# Breadth-first search

http://joseph-harrington.com/2012/02/breadth-first-search-visual/



*find the shortest path between two nodes in a graph*

# Graph representations

# Adjacency matrices

A graph with $n$ nodes can be represented by an $n \times n$ square matrix $M$.

Matrix element $c_{ij} > 0$ indicates an edge from node $n_i$ to $n_j$.

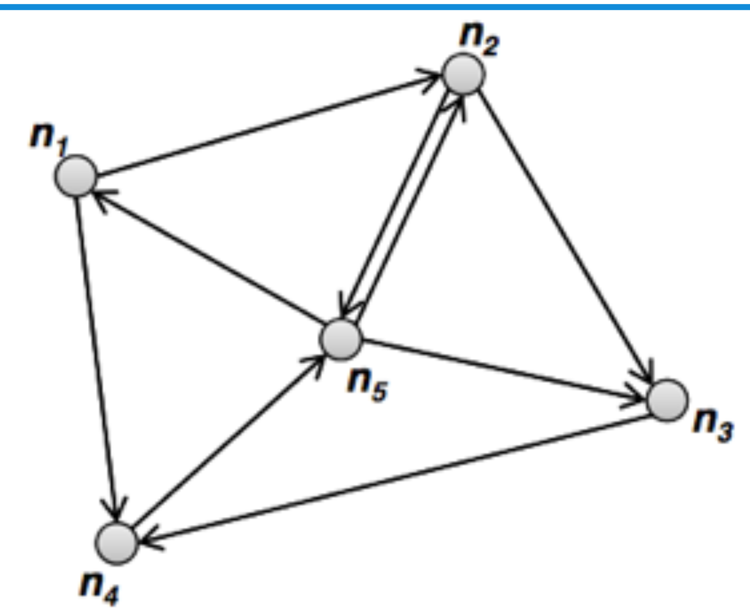
- Edges in **unweighted** graphs: 1 (edge exists), 0 (no edge exists)
- Edges in **weighted** graphs: matrix contains edge weights
- **Undirected** graphs use half the matrix
- **Advantage**: mathematically easy manipulation
- **Disadvantage**: space requirements

# Adjacency matrices

A graph with $n$ nodes can be represented by an $n \times n$ square matrix $M$.

Matrix element $c_{ij} > 0$ indicates an edge from node $n_i$ to $n_j$.

- Edges in **unweighted** graphs
- Edges in **weighted** graphs: r
- **Undirected** graphs use half t




| | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ |
|---|---|---|---|---|---|
| $n_1$ | 0 | 1 | 0 | 1 | 0 |
| $n_2$ | 0 | 0 | 1 | 0 | 1 |
| $n_3$ | 0 | 0 | 0 | 1 | 0 |
| $n_4$ | 0 | 0 | 0 | 0 | 1 |
| $n_5$ | 1 | 1 | 1 | 0 | 0 |

adjacency matrix

- **Advantage**: mathematically easy manipulation
- **Disadvantage**: space requirements

# Adjacency list

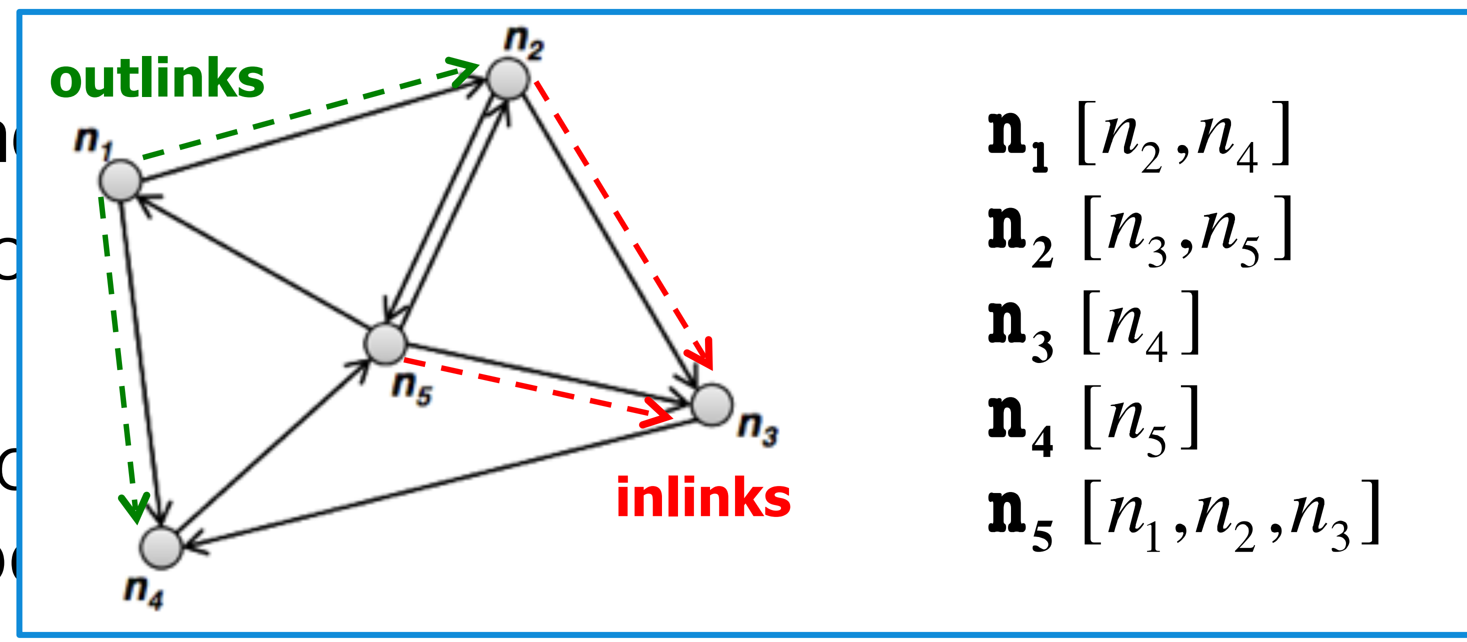
- A much more **compressed** representation
  - On sparse graphs
- Only edges **that exist** are encoded in adjacency lists
- Two options to encode **undirected** edges:
  - Encode each edge twice (the nodes appear in each other's adjacency list)
  - Impose an order on nodes and encode edges only on the adjacency list of the node that comes first in the ordering
- **Disadvantage**: some graph operations are more difficult compared to the matrix representation

# Adjacency list

- A much more **compressed** representation
    - On sparse graphs

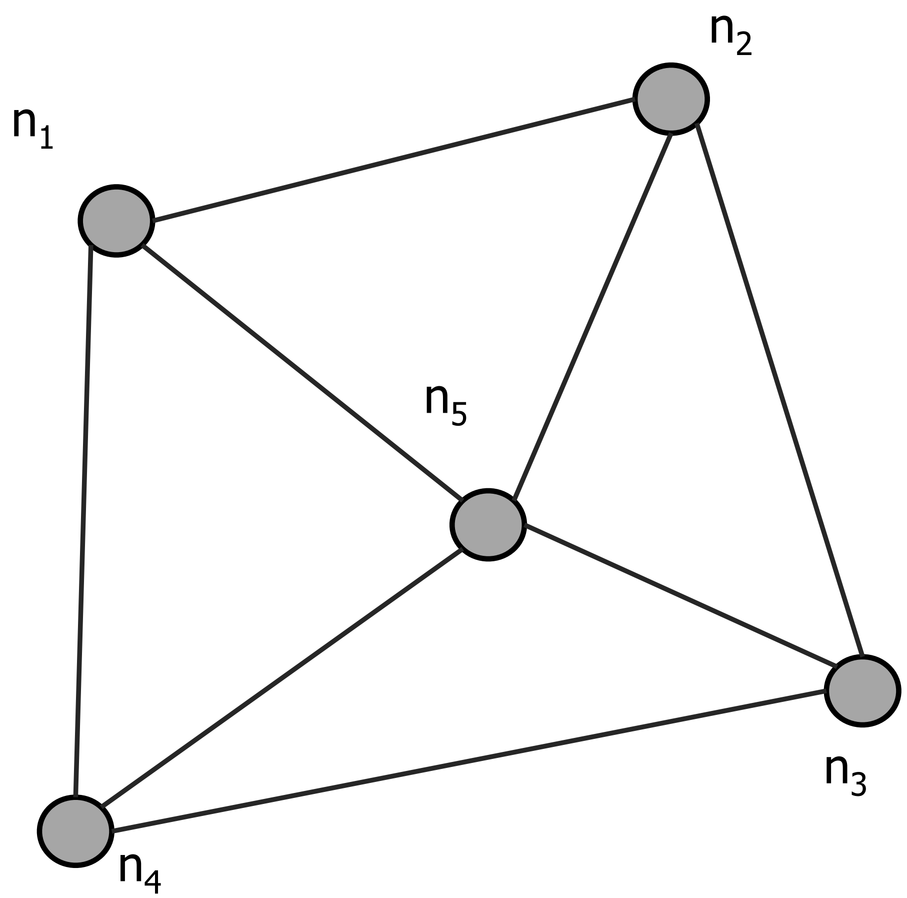
- Only edges **that exist** are encoded in adjacency lists

- Two options to encode **un**
    - Encode each edge twic
      adjacency list)
    - Impose an order on noc
      adjacency list of the noc




- **Disadvantage**: some graph operations are more difficult compared to the matrix representation

# Adjacency list




each edge twice

$\mathbf{n}_1$ $[n_2, n_4, n_5]$

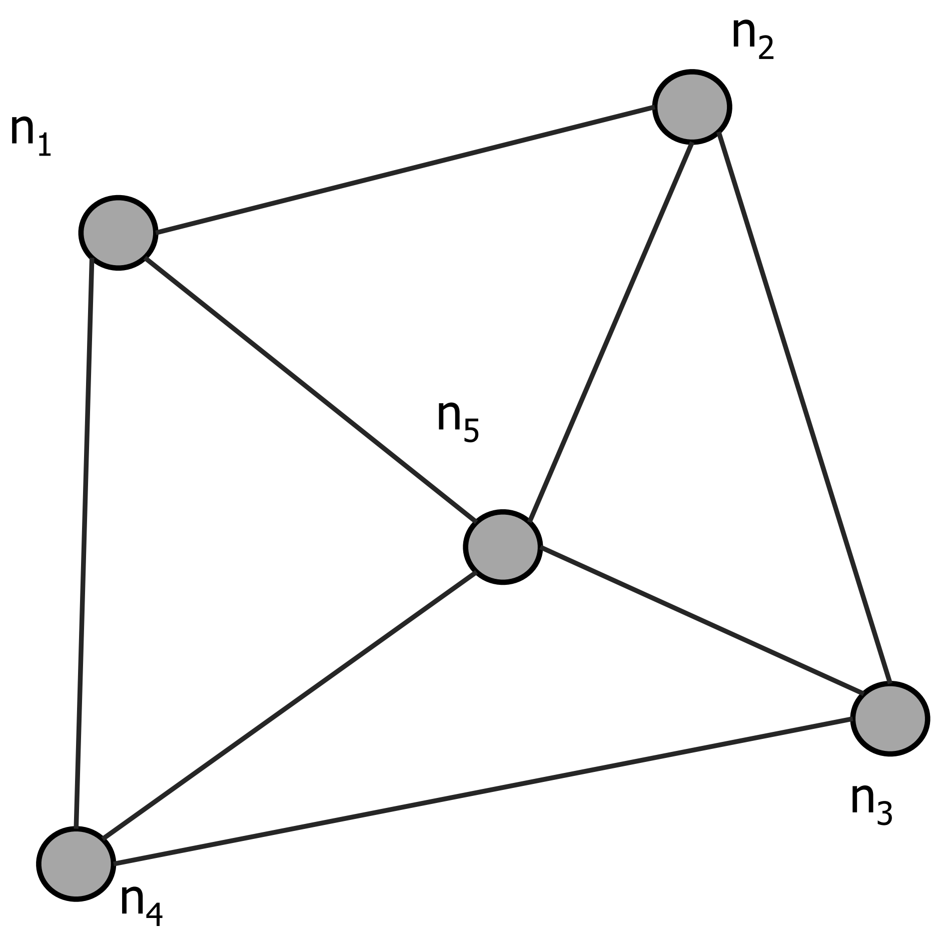
$\mathbf{n}_2$ $[n_1, n_3, n_5]$

$\mathbf{n}_3$ $[n_2, n_4, n_5]$

$\mathbf{n}_4$ $[n_1, n_3, n_5]$

$\mathbf{n}_5$ $[n_1, n_2, n_3, n_4]$

# Adjacency list




node ordering

$\mathbf{n_1}$ $[n_2, n_4, n_5]$

$\mathbf{n_2}$ $[n_3, n_5]$

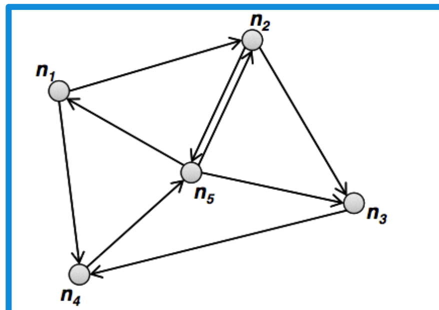
$\mathbf{n_3}$ $[n_4, n_5]$

$\mathbf{n_4}$ $[n_5]$

$\mathbf{n_5}$ $[]$

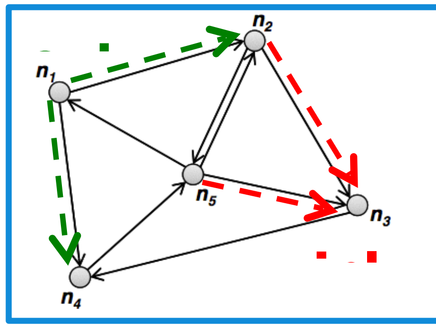
# Adjacency matrices vs. lists

- A less compressed representation (matrix) makes some computations easier




| | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ |
|---|---|---|---|---|---|
| $n_1$ | 0 | 1 | 0 | 1 | 0 |
| $n_2$ | 0 | 0 | 1 | 0 | 1 |
| $n_3$ | 0 | 0 | 0 | 1 | 0 |
| $n_4$ | 0 | 0 | 0 | 0 | 1 |
| $n_5$ | 1 | 1 | 1 | 0 | 0 |

adjacency matrix




$\mathbf{n}_1$ $[n_2, n_4]$
$\mathbf{n}_2$ $[n_3, n_5]$
$\mathbf{n}_3$ $[n_4]$
$\mathbf{n}_4$ $[n_5]$
$\mathbf{n}_5$ $[n_1, n_2, n_3]$

- Computing **inlinks**
  - Matrix: scan the column and count
  - List: difficult, worst case all data needs to be scanned

- Computing **outlinks**
  - Matrix: scan the rows and count
  - List: outlinks are natural

# Breadth-first search
# (in detail)

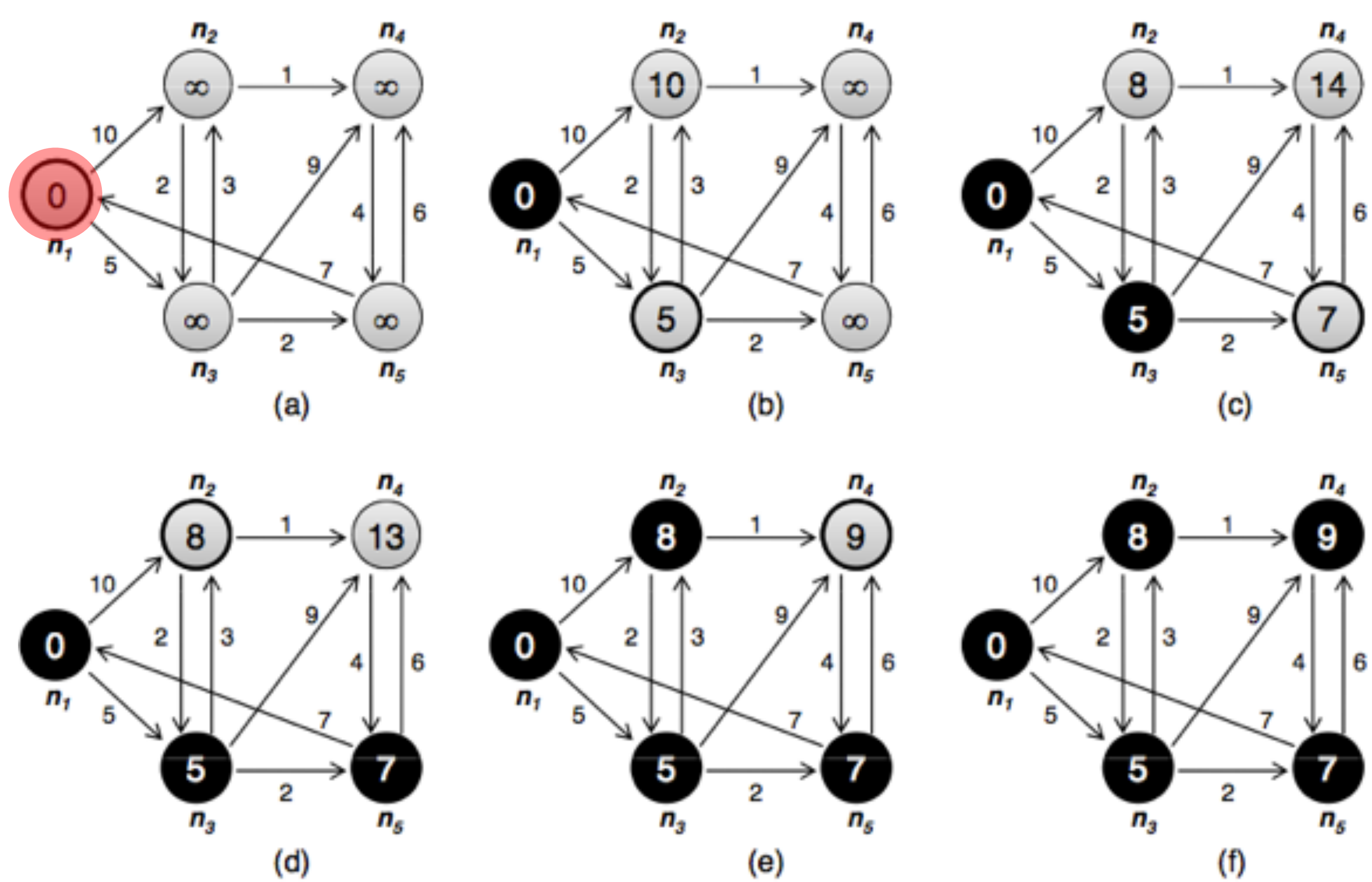
# Single-source shortest path

## *Standard solution: Dijkstra’s algorithm*

**Task**: find the shortest path from a **source node** to all other nodes in the graph

In each step,
find the minimum
edge of a node
not yet visited.

6 iterations.




Source: **Data-Intensive Text Processing with MapReduce**

# Single-source shortest path

## *Standard solution: Dijkstra's algorithm*

**Task**: find the shortest path from a **source node** to all other nodes in the graph

**Input**:
-directed connected graph in adjacency list format
-edge distances in $w$
-source $s$

$\textsc{Dijkstra}(G, w, s)$
    $d[s] \leftarrow 0$    **source node**
    **for all** vertex $v \in V$ **do**
        $d[v] \leftarrow \infty$
**starting distance: infinite for all nodes**
    $Q \leftarrow \{V\}$
    **while** $Q \neq \emptyset$ **do**    Q is a **global priority queue** sorted by current distance
        $u \leftarrow \textsc{ExtractMin}(Q)$
        **for all** vertex $v \in u.\textsc{AdjacencyList}$ **do**
            **if** $d[v] > d[u] + w(u, v)$ **then**
                $d[v] \leftarrow d[u] + w(u, v)$    adapt distances

Source: **Data-Intensive Text Processing with MapReduce**

# Single-source shortest path
## In the MapReduce world: parallel BFS

**Task**: find the shortest path from a **source node** to all other nodes in the graph.

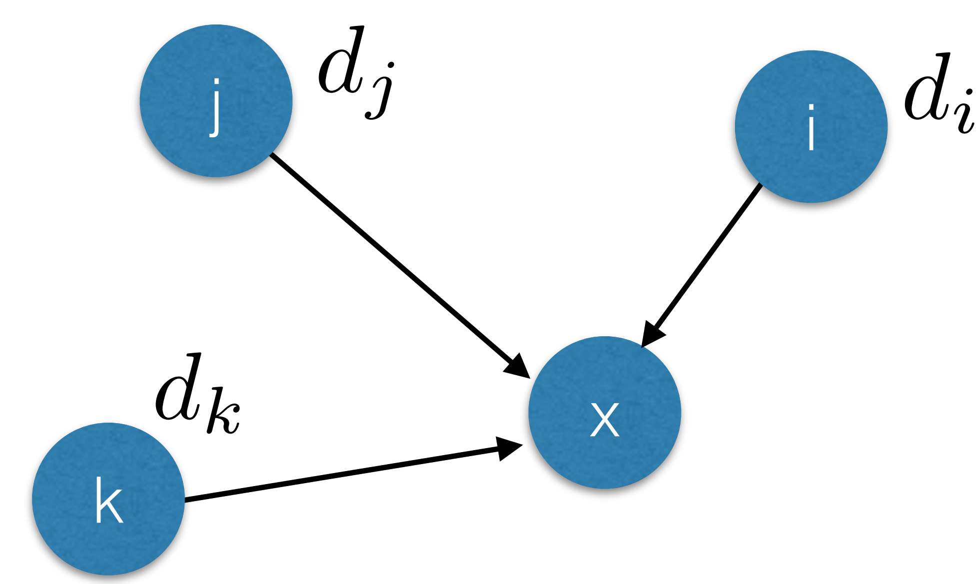
- **Brute force approach**: parallel breadth-first search
- **Intuition**:
  - Distance of all nodes N directly connected to the source is one
  - Distance of all nodes directly connected to nodes in N is two
  - …
  - Multiple path to a node x: the shortest path must go through one of the nodes having an outgoing edge to x; use the minimum

**Here: edges have unit weight.**

# Single-source shortest path
## In the MapReduce world: parallel BFS

**Task**: find the shortest path from a **source node** to all other nodes in the graph.

- **Brute force approach**: parallel breadth-first search
- **Intuition**:
  - Distance of all nodes N
  - Distance of all nodes dir
  - …
  - Multiple path to a node x one of the nodes having minimum




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

**Here: edges have unit weight.**

# Single-source shortest path

## In the MapReduce world: parallel BFS

**Mapper**: emit all distances, and the graph structure itself

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

**Reducer**: update distances and emit the graph structure

```
class Reducer
    method Reduce(nid m, [d_1, d_2, ...])
        d_min ← ∞
        M ← ∅
        for all d ∈ counts [d_1, d_2, ...] do
            if IsNode(d) then
                M ← d                               ▷ Recover graph structure
            else if d < d_min then                  ▷ Look for shorter distance
                d_min ← d
        M.Distance ← d_min                          ▷ Update shortest distance
        Emit(nid m, node M)
```

Source: **Data-Intensive Text Processing with MapReduce**



**Edges have unit weight.**

# Single-source shortest path

## In the MapReduce world: parallel BFS

**Mapper**: emit all distances, and the graph structure itself

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

**Reducer**: update distances and emit the graph structure

```
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)
```

Overloading of value type: distance (int) or complex data structure.

In practice: wrapper class with indicator variable.

Source: **Data-Intensive Text Processing with MapReduce**

**Edges have unit weight.**

# Single-source shortest path
## In the MapReduce world: parallel BFS

- Each **iteration** of the algorithm is **one MapReduce 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. …

- Between iterations (jobs) the **graph structure needs to be passed along**
  - Reducer output is input for the next iteration (job)



**Edges have unit weight.**

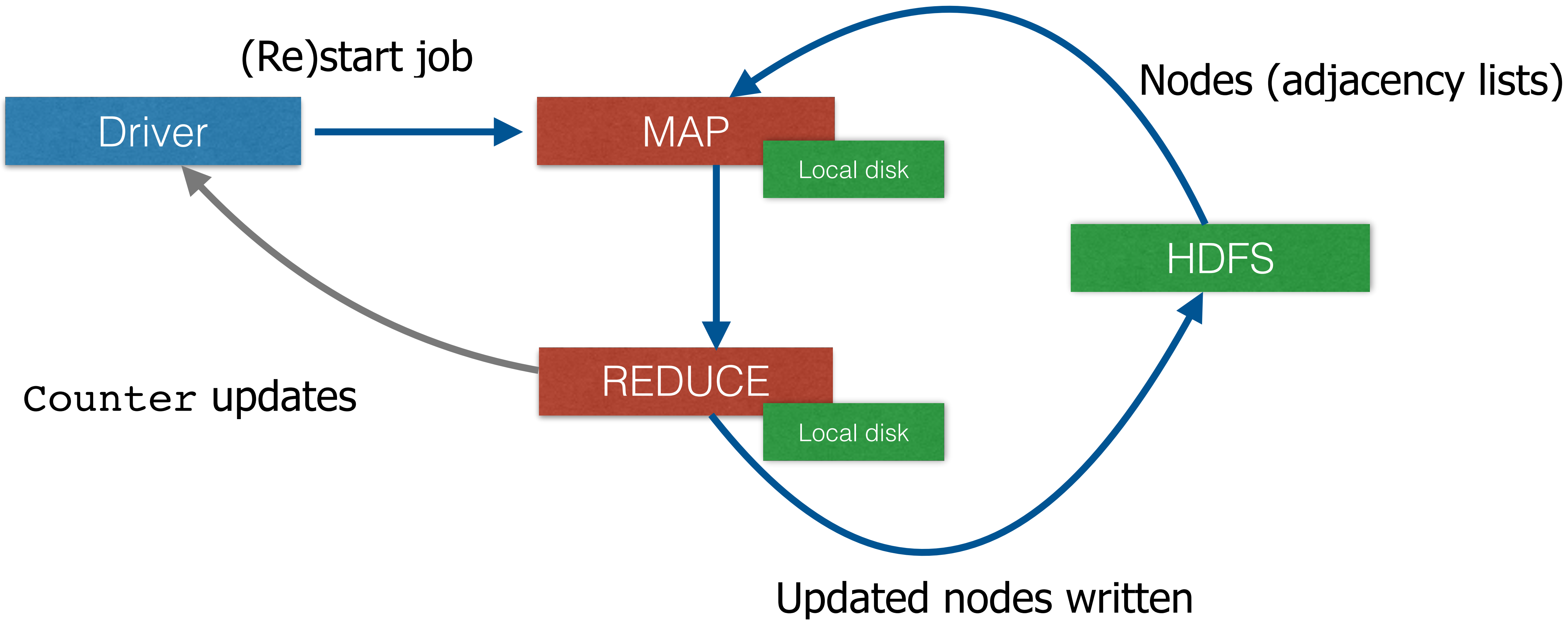
# Single-source shortest path
## In the MapReduce world: parallel BFS

- 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”- Milgram)
- 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
- Emit current shortest paths in the Mapper as well



**Edges have unit weight.**

# Single-source shortest path
## In the MapReduce world: parallel BFS



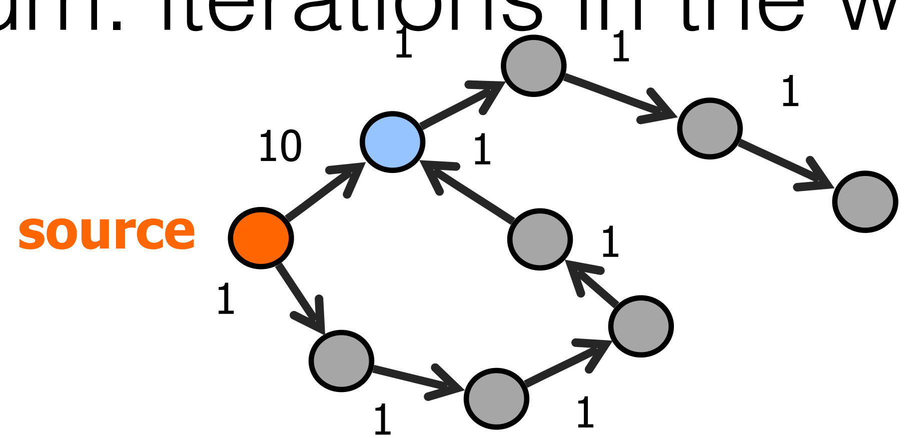

**Disadvantage: a lot of reading and writing to/from HDFS**

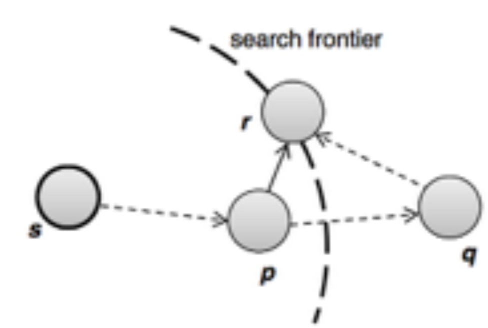
# Single-source shortest path
## In the MapReduce world: parallel BFS

**Task**: find the shortest path from a source node to all other nodes when edges **have positive distances > 1**

- **Two changes required**:
  - 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 vs. parallel BFS

- **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

in general …

# 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.)

- **MapReduce 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

# The Web graph

**Open Directory Project**:
5,007,664 sites
94,441 editors
over 1,010,258 categories

# The Web

- **Vannevar Bush** envisioned hypertext in the 1940's
- First **hypertext** systems were created in the 1970's
- The World Wide Web was formed in the early 1990's
    - Creator: Tim Berners-Lee
    - Make documents easily available to anyone (Web pages)
    - Easy access to such Web pages using a browser
- Early Web years
    - Full-text search engines (Altavista, Excite and Infoseek) vs.
    - Taxonomies populated with pages in categories (ODP, Yahoo! Directory)

# The Web

- Nearly impossible to discover content without search engines
  - Estimating the size of the Web is a research area by itself
  - Indexed Web has **billions** of pages
  - **Deep Web**

- Users view the Web through the lense of the search engine

- Pages not indexed (or ranked at low positions) by search engines are unlikely to be found by users

# Graph structure in the Web

Broder et al., 1999

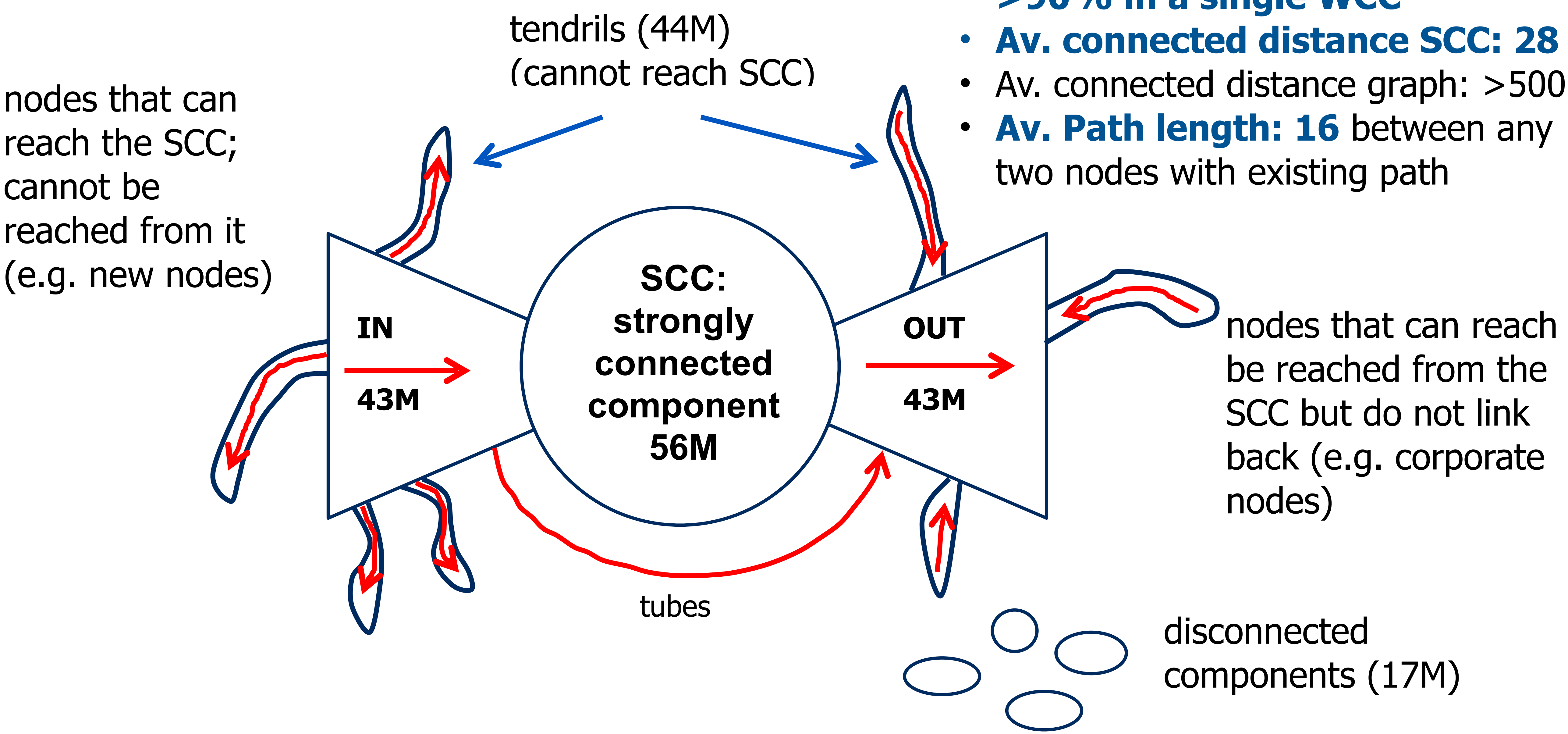
- Insights important for:
  - **Crawling strategies**
  - **Understanding** the sociology of content creation
  - **Analyzing** the behaviour of algorithms that rely on link information (e.g. HITS, PageRank)
  - Predicting the **evolution** of web structures
  - Predicting the **emergence** of new phenomena in the Web graph

- Data: Altavista crawl from 1999 with **200 million pages** and **1.5 billion links**

# The Web as a "bow tie"

Broder et al., 1999

- ~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



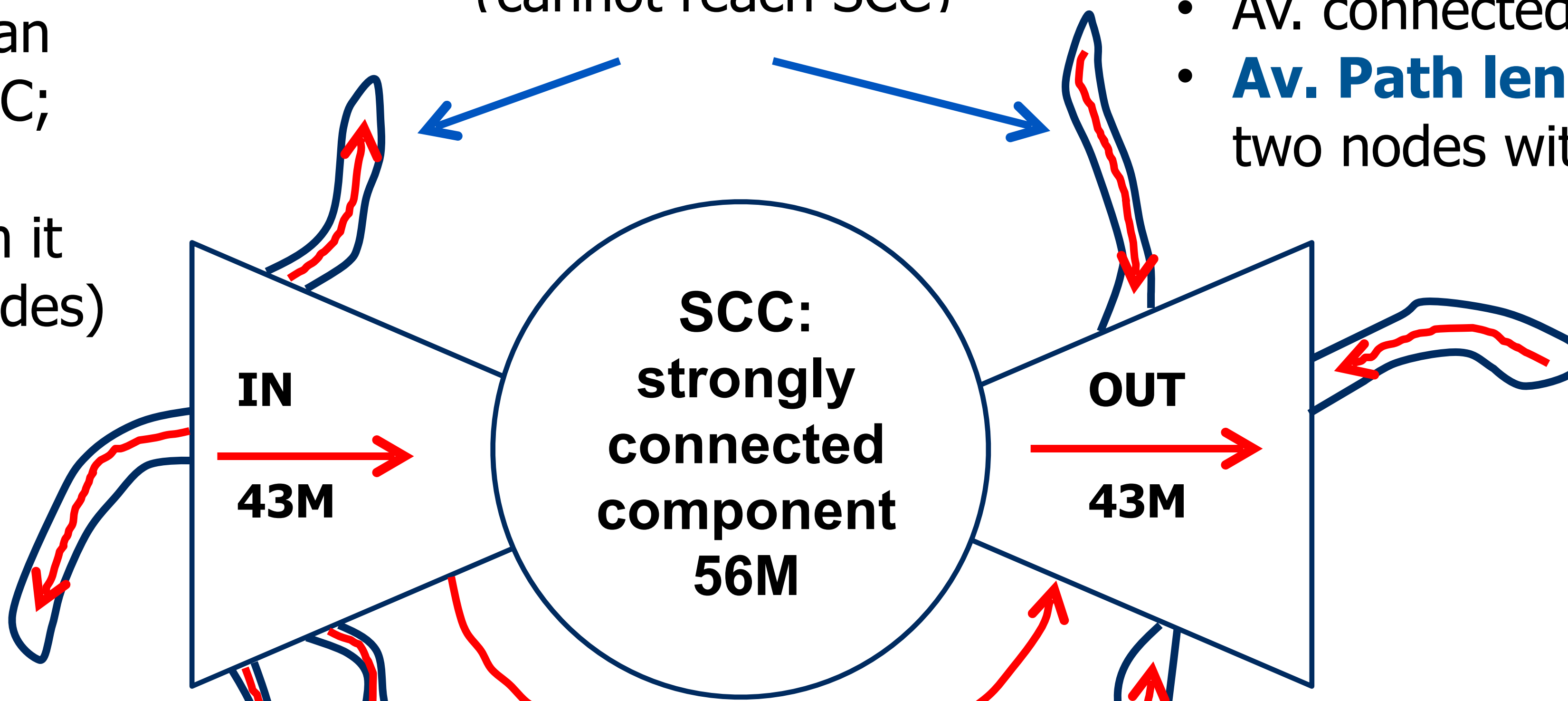
# The Web as a “bow tie”

Broder et al., 1999

- ~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

tendrils (44M)
(cannot reach SCC)

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




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

“In a sense the web is much like a complicated organism, in which the local structure at a microscopic scale looks very regular like a biological cell, but the global structure exhibits interesting morphological structure (body and limbs) that are not obviously evident in the local structure.”

# PageRank

Page et al., 1998

- 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


Page et al., 1998

- **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

- A page **distributes** "importance" through its outlinks

- Simple PageRank (iteratively):

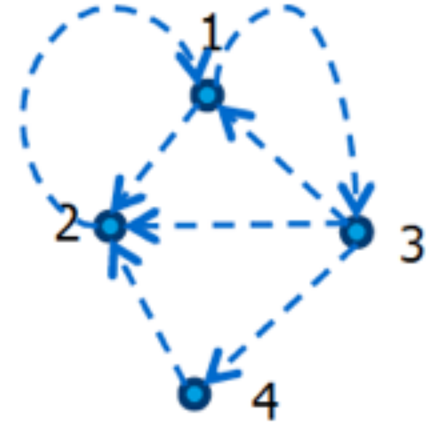
out-degree of node $u$

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

all nodes linking to $v$

# PageRank

## Simplified formula




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

outlinks of node 1

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

*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**

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

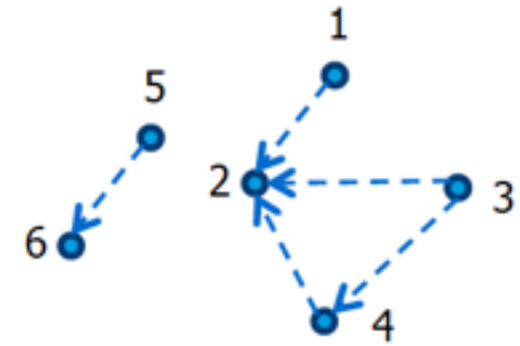
$$W^{17} \times \vec{R}' = \begin{pmatrix} 0.40 \\ 0.34 \\ 0.20 \\ 0.07 \end{pmatrix}$$

**Random surfer model:**

- Probability that a random surfer starts at a random page and ends at page `px`
- A random surfer at a page with 3 outlines randomly picks one (1/3 prob.)

# PageRank

## Reality




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

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

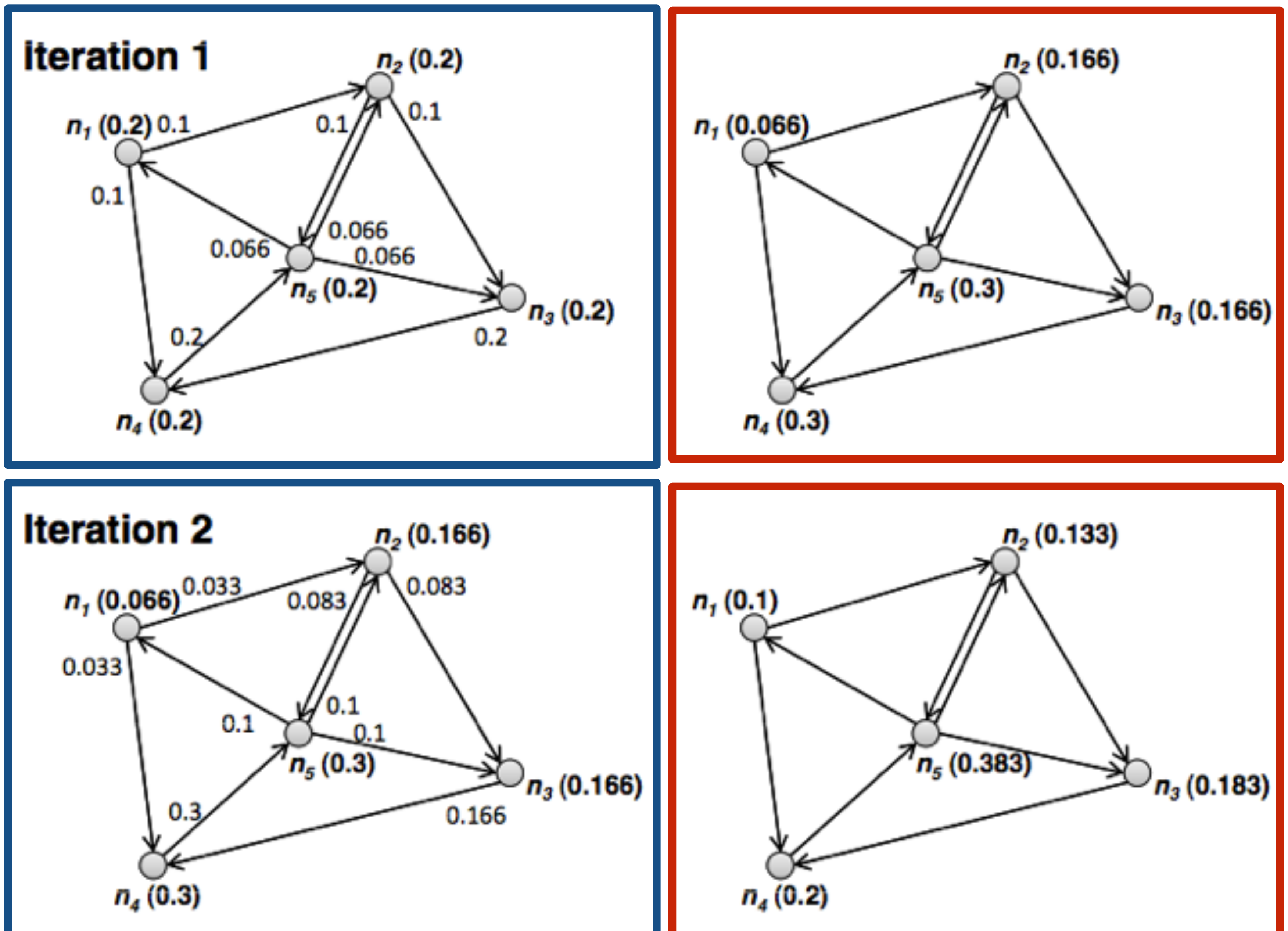
## An informal sketch

- 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

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






Source: **Data-Intensive Text Processing with MapReduce**

# PageRank in MapReduce

## Pseudocode: simplified PageRank

```
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)
```



Source: **Data-Intensive Text Processing with MapReduce**

# PageRank in MapReduce

## Jump factor and “dangling” nodes

- **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)

One iteration of PageRank requires two MR jobs!

# 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

Warning: on today's Web, PageRank requires additional modifications (spam, spam, spam)

# Graph processing notes

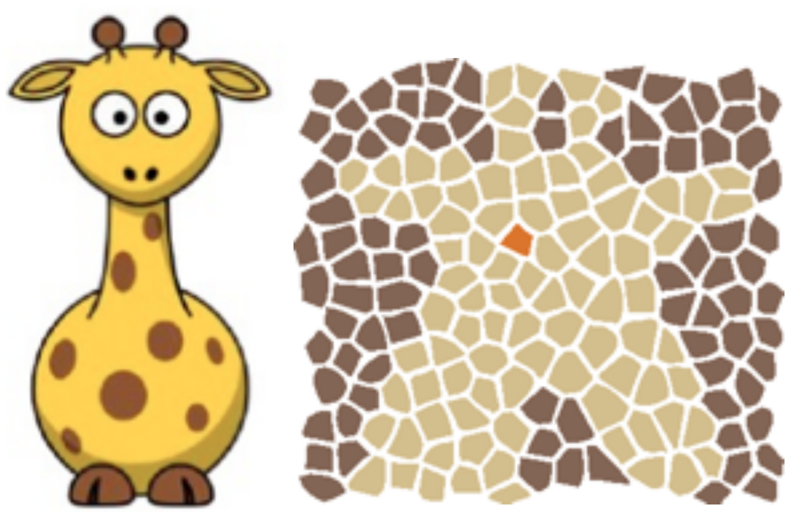
- In **dense graphs**, MR running time would be dominated by the shuffling of the intermediate data across the network
  - **Worst case**: $O(n^2)$
  - **Impractical** for MR (commodity hardware)
- Often, combiners and in-mapper combining patterns can be used to speed up the process
- **Data localization** can be **difficult**
  - Combiners are only useful if there is something to aggregate (e.g. for PR several nodes pointing to the same target in a single MAPPER)
  - Heuristics: e.g. pages from the same domain to the same MAPPER

# 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
  - More details in the next lecture



# Summary

- Graph problems in the real world
- A bit of graph theory
- Adjacency matrices vs. adjacency lists
- Breadth-first search
- PageRank

# References


- **Data-Intensive Text Processing with MapReduce** by Jimmy Lin and Chris Dyer. Chapter 5.
- **Graph structure in the Web.** Broder et al. 1999.
- **The PageRank Citation Ranking: Bringing Order to the Web**. Page et al. 1999.

THE END