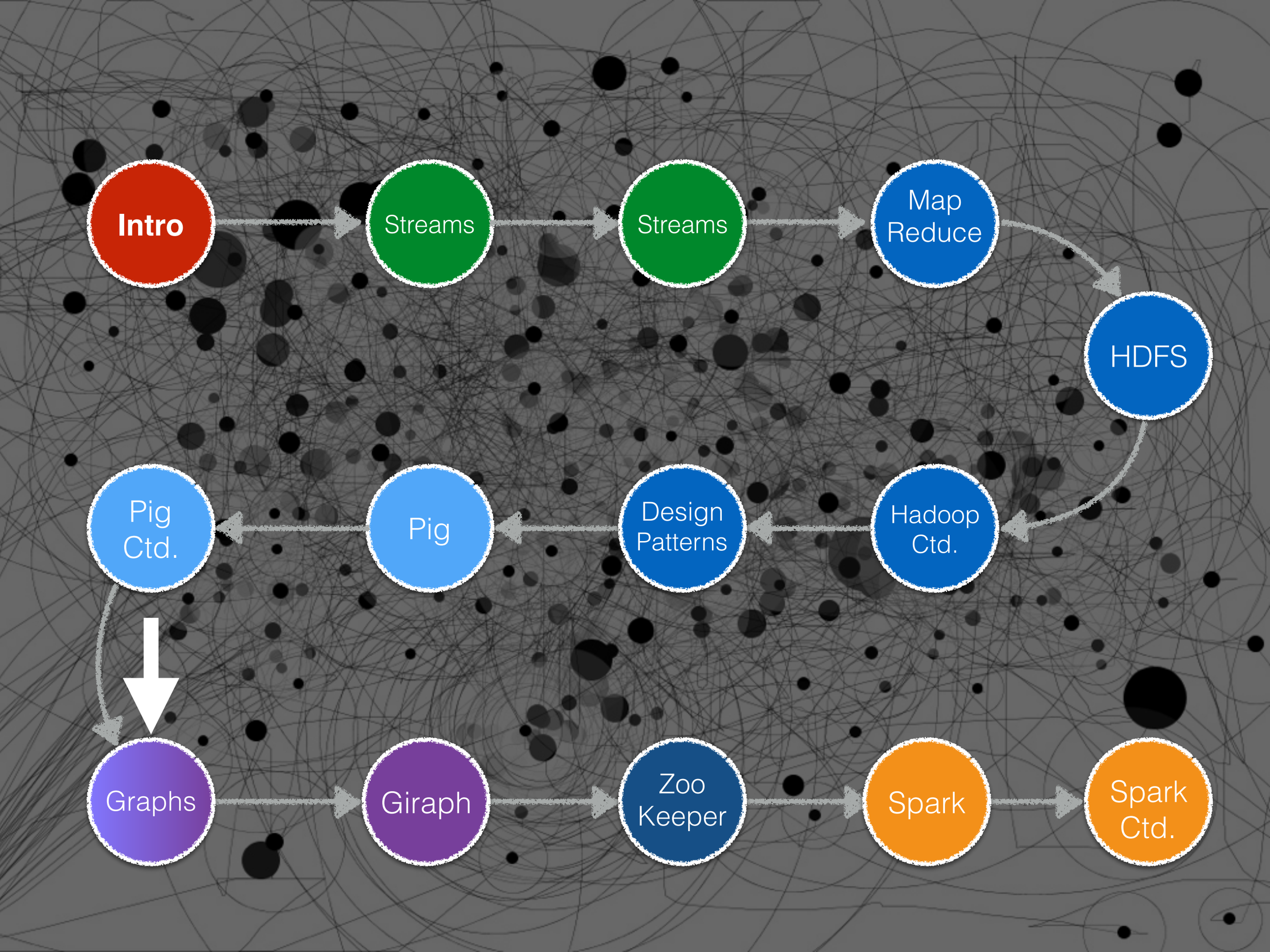
TI2736-B

Big Data Processing

Claudia Hauff
ti2736b-ewi@tudelft.nl

TU Delft


Intro
Streams
Streams
Map Reduce
HDFS
Hadoop Ctd.
Design Patterns
Pig
Pig Ctd.
Graphs
Giraph
Zoo Keeper
Spark
Spark Ctd.

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

- **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

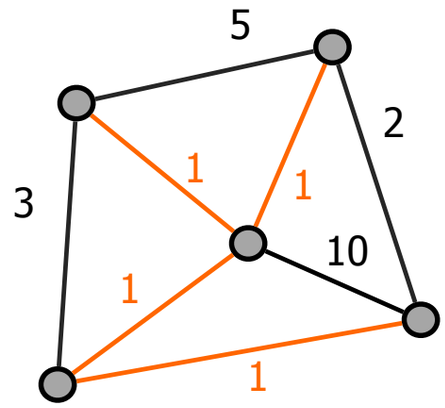
- **Graph search**
  - Friend recommend. in social networks
  - Expert finding in social networks

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

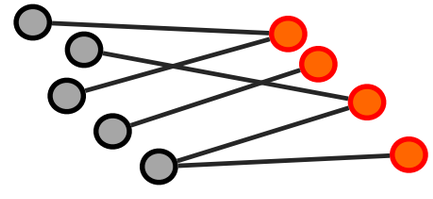
- **Graph clustering**
  - Sub-communities in large graphs

Real-world problems to solve

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



- **Bipartite graph matching**: match 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.

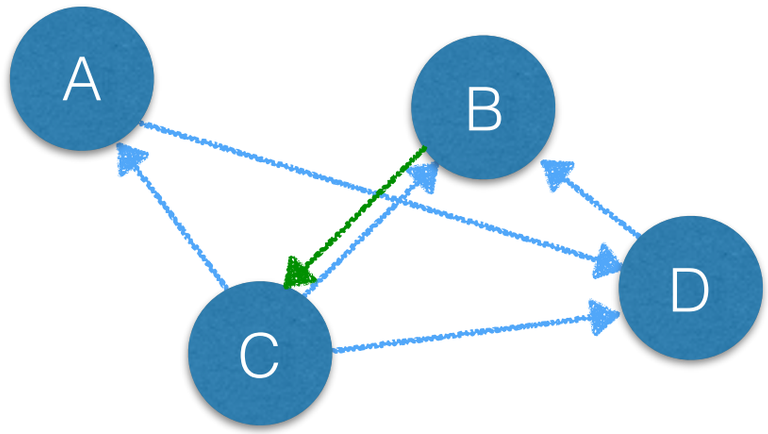
Real-world graphs are often **sparse**: the number of actual edges is **far smaller** than the number of possible edges.

flow capacity constraints

A bit of graph theory

Connected components

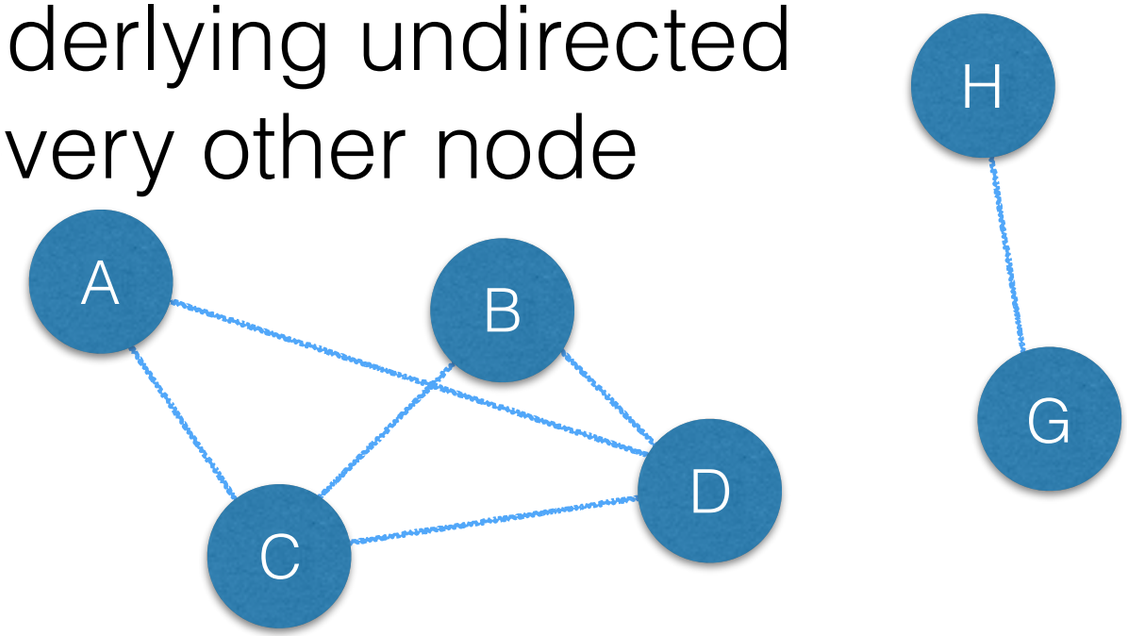
- **Strongly** connected component (SCC): directed graph with a path from each node to every other node



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

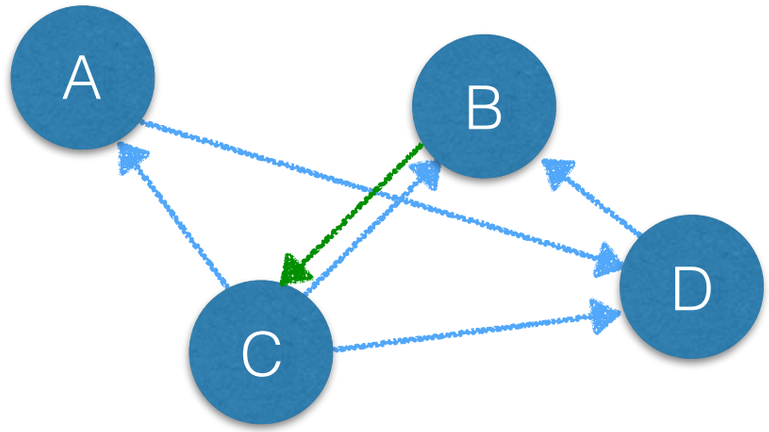
strongly connected

- **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

Graph notation



graph $G = (V, E)$

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

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

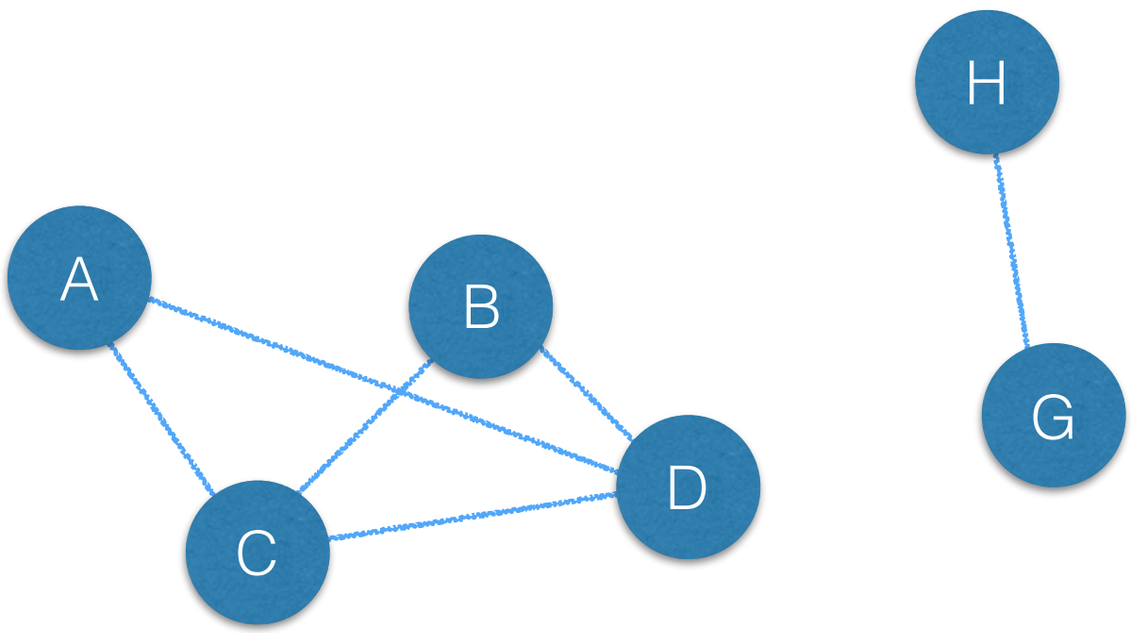
shortest distance between 2 nodes $d(A, B) = 2, d(C, B) = 1, d(A, C) = 3$

Graph notation

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

undirected edges

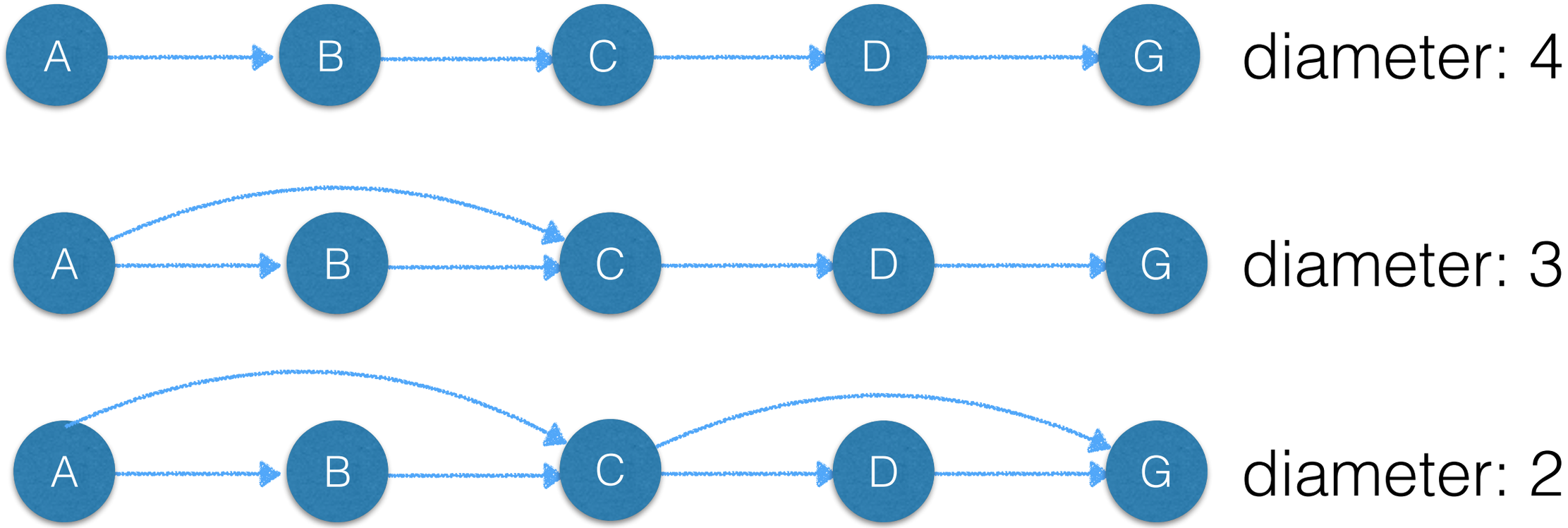
infinite distance



Graph diameter

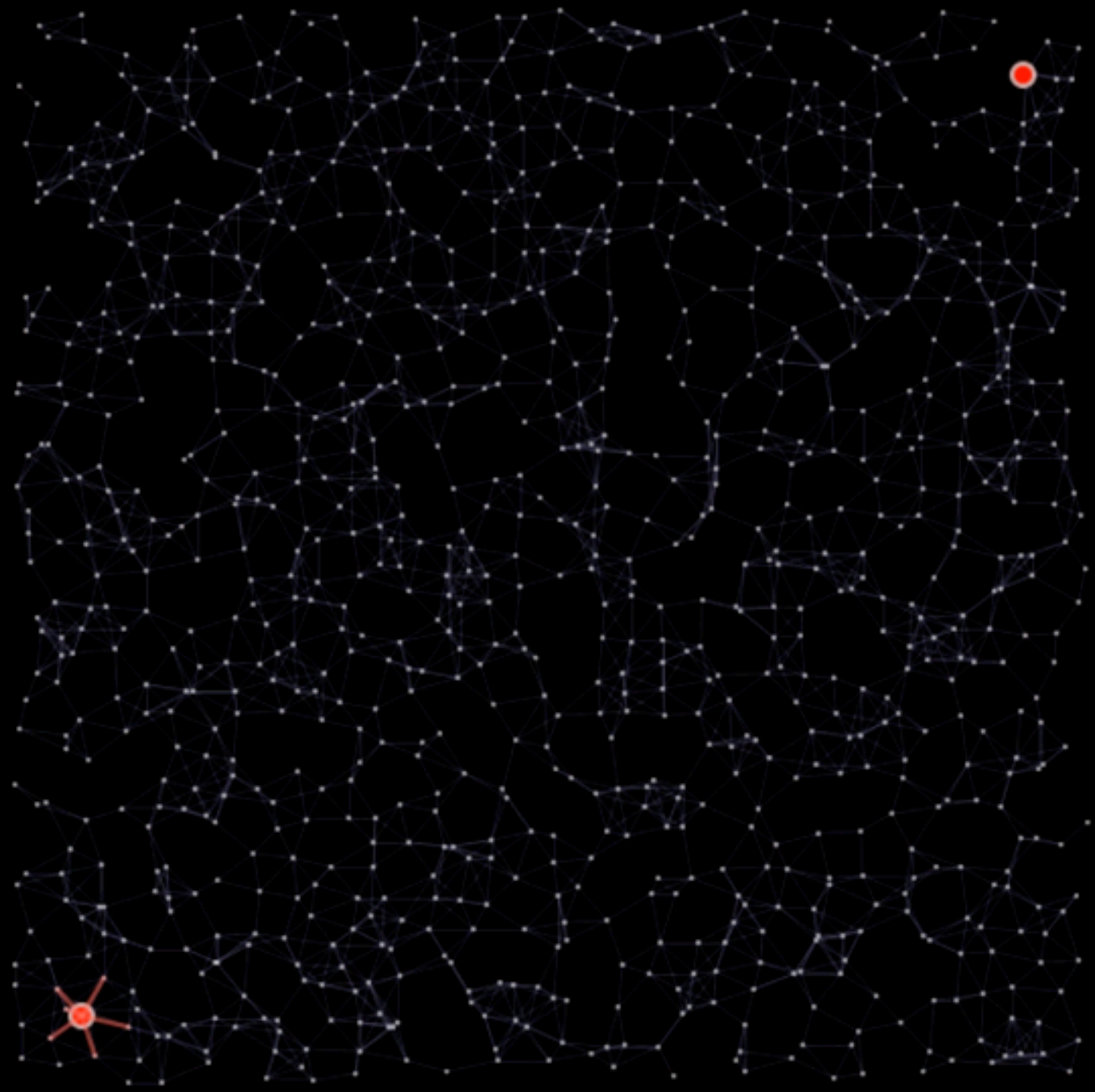
Definition: **longest shortest path** in the graph

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



Breadth-first search

find the shortest path between two nodes in a graph



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

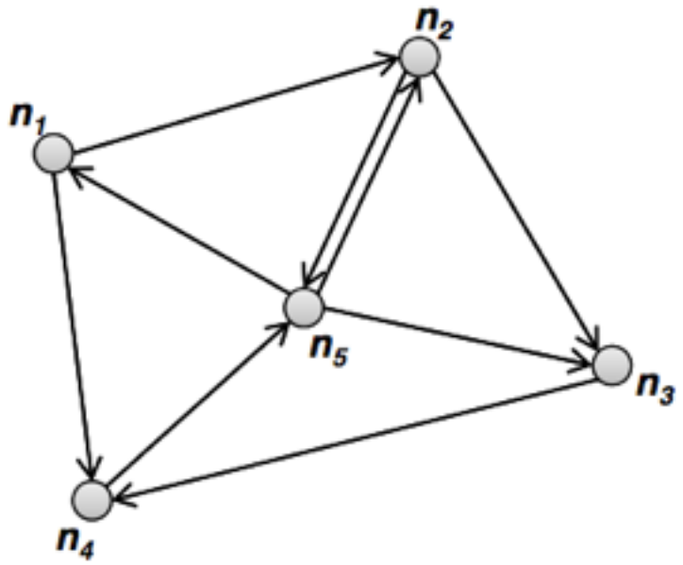
Graph representations

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



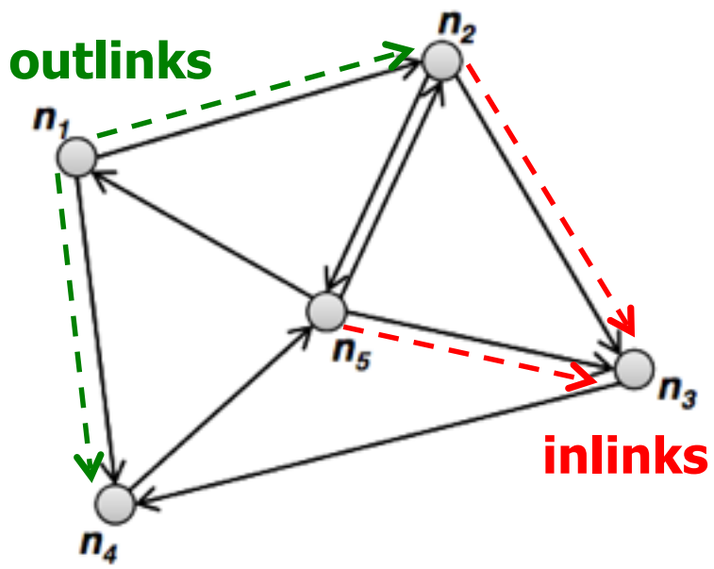
	n_1	n_2	n_3	n_4	n_5
n_1	0	1	0	1	0
n_2	0	0	1	0	1
n_3	0	0	0	1	0
n_4	0	0	0	0	1
n_5	1	1	1	0	0

adjacency matrix

Adjacency list

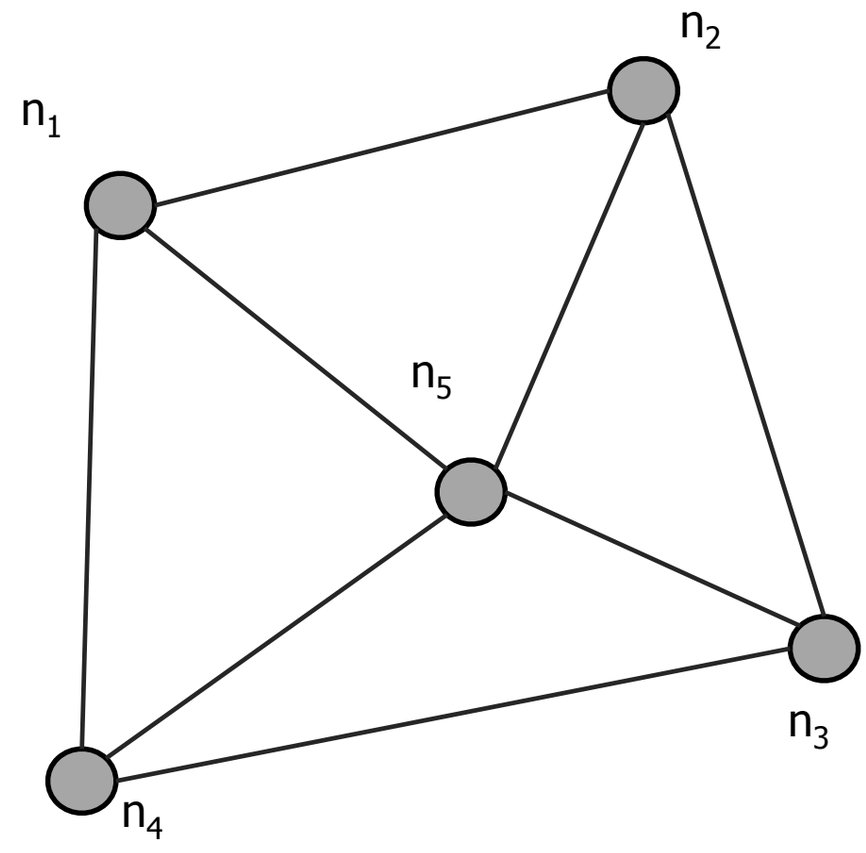
- A much more **compressed** representation (for sparse graphs)
- Only edges **that exist** are encoded in adjacency lists
- Two options to encode **undirected** edges:
  - Encode **each edge twice** (the nodes appear in each other's adjacency list)
  - **Impose an order** on nodes and encode edges only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

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



$\mathbf{n_1}\ [n_2, n_4]$
$\mathbf{n_2}\ [n_3, n_5]$
$\mathbf{n_3}\ [n_4]$
$\mathbf{n_4}\ [n_5]$
$\mathbf{n_5}\ [n_1, n_2, n_3]$

Adjacency list



each edge twice

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

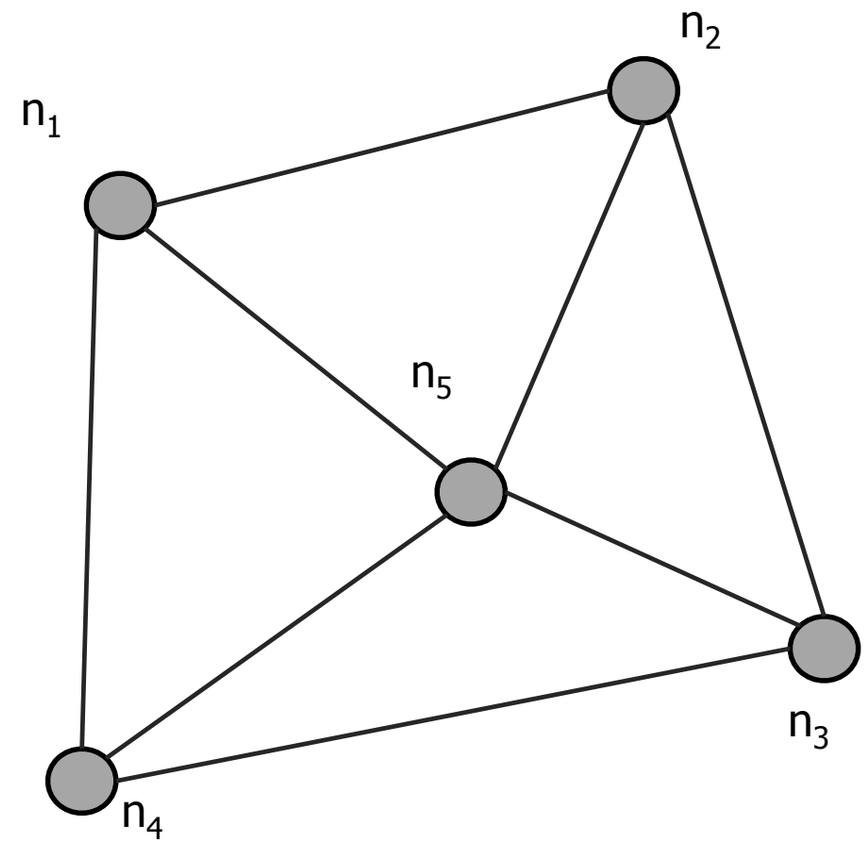
$\mathbf{n_2}\ [n_1, n_3, n_5]$

$\mathbf{n_3}\ [n_2, n_4, n_5]$

$\mathbf{n_4}\ [n_1, n_3, n_5]$

$\mathbf{n_5}\ [n_1, n_2, n_3, n_4]$

Adjacency list



node ordering

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

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

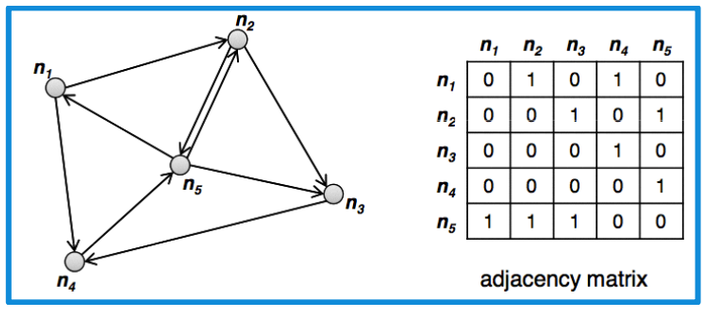
$\mathbf{n_3}\ [n_4, n_5]$

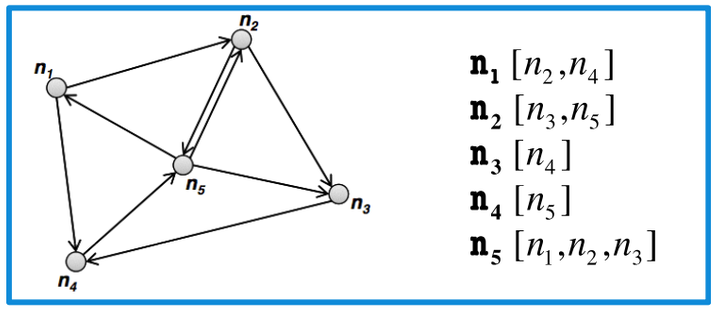
$\mathbf{n_4}\ [n_5]$

$\mathbf{n_5}\ []$

Adjacency matrices vs. lists

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





- 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

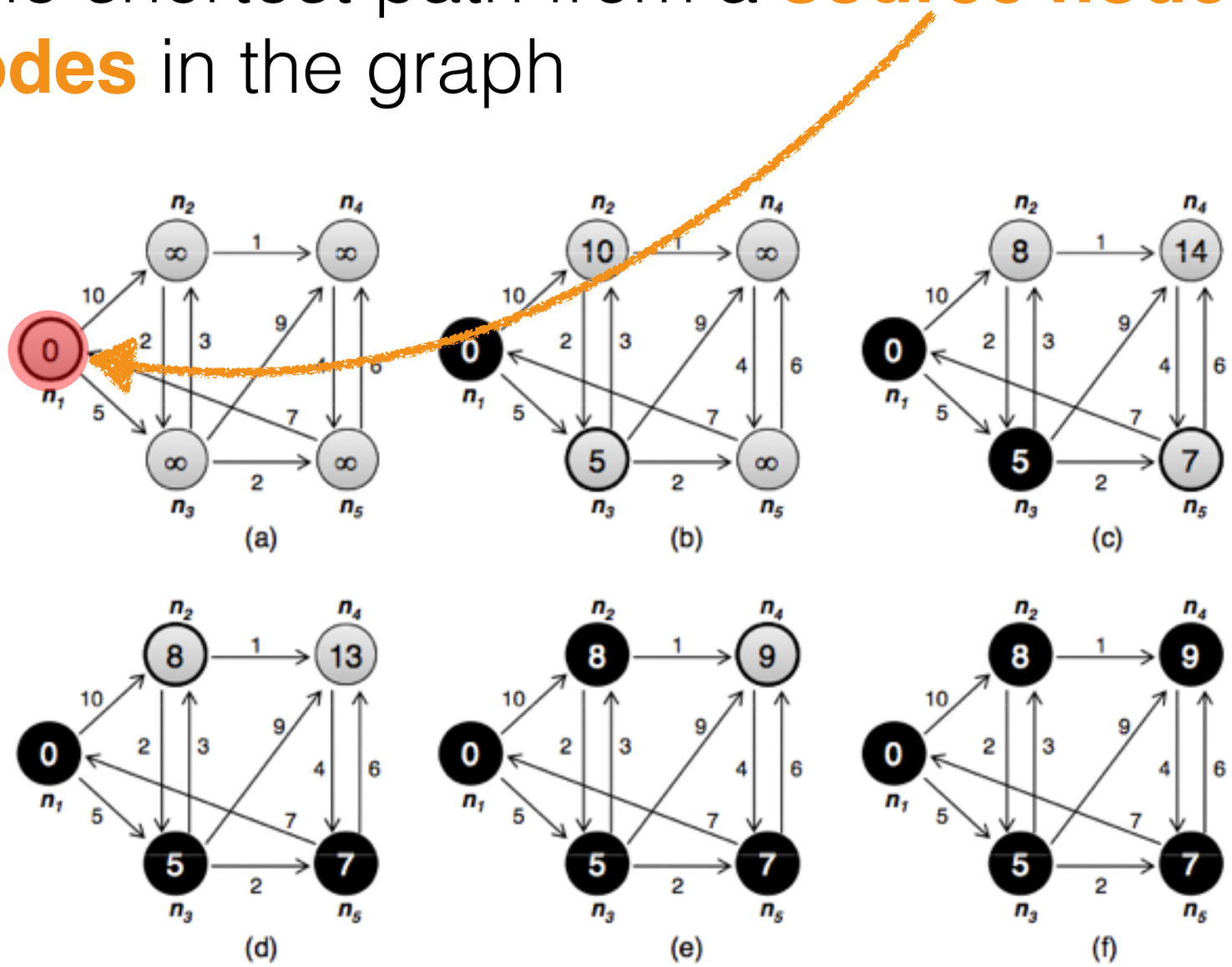
Breadth-first search (in detail)

Single-source shortest path

Standard solution: Dijkstra's algorithm I

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.



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

Single-source shortest path

Standard solution: Dijkstra's algorithm II

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

```
1:  DIJKSTRA(G, w, s)
2:      d[s] ← 0                          source node
3:      for all vertex v ∈ V do
4:          d[v] ← ∞
5:      Q ← {V}
6:      while Q ≠ ∅ do
7:          u ← EXTRACTMIN(Q)
8:          for all vertex v ∈ u.ADJACENCYLIST do
9:              if d[v] > d[u] + w(u, v) then
10:                 d[v] ← d[u] + w(u, v)
```

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

starting distance: infinite for all nodes

Q is a **global priority queue sorted by current distance**

adapt distances

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

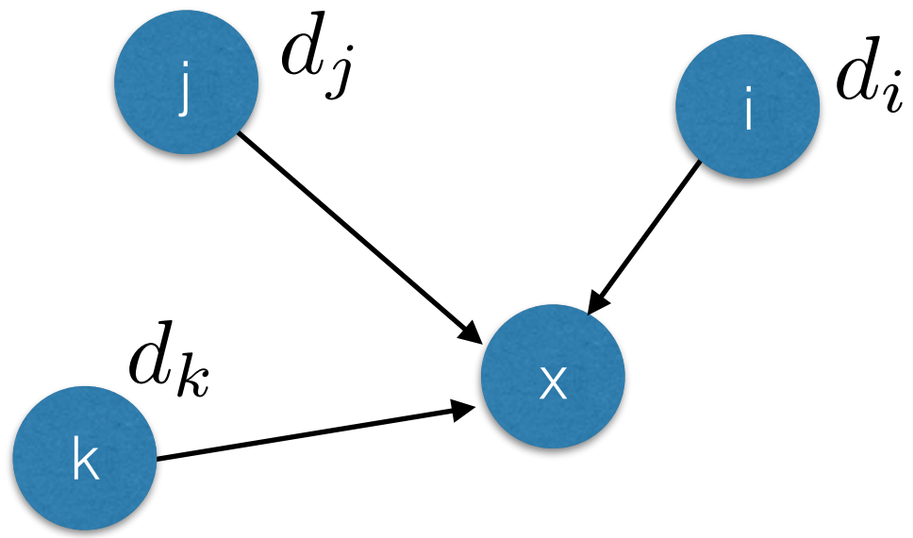
Single-source shortest path

In the MapReduce world: parallel BFS I

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**



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

1 Hadoop job per iteration.

Single-source shortest path

In the MapReduce world: parallel BFS II

```
Node {
    distance
    adjacencyList
}
```

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

Mapper: emit all distances, and the graph structure itself

```
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
8:             else if d < d_min then
9:                 d_min ← d
10:        M.Distance ← d_min
11:        Emit(nid m, node M)
```

Reducer: update distances and emit the graph structure

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 III

- 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. …

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

Edges have unit weight.

Single-source shortest path

In the MapReduce world: parallel BFS IV

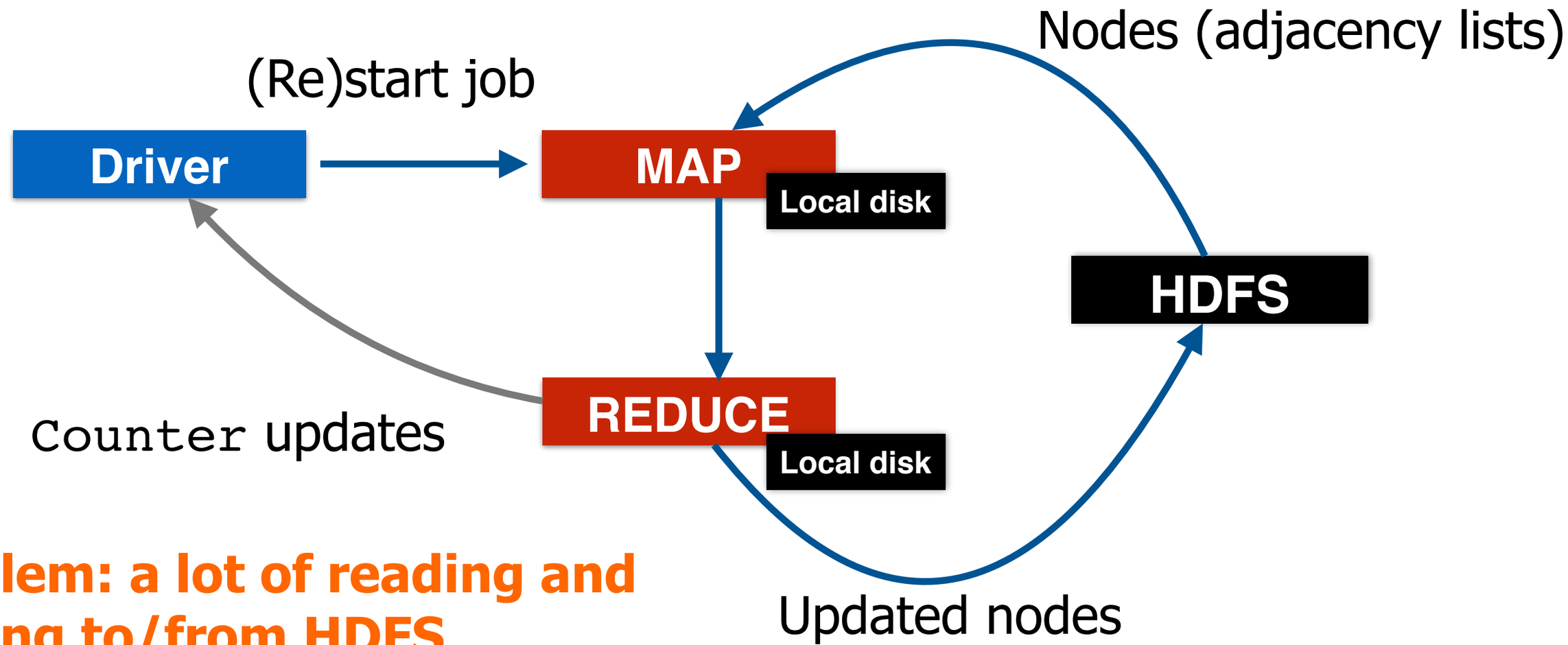
- How many iterations are necessary to compute the shortest path to all nodes?
  - **Diameter** of the graph (greatest distance between a pair of nodes)
    - Diameter is **usually small** (“six degrees of separation”)
- In **practice**: iterate until all node distances are less than +infinity
  - Assumption: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

Edges have unit weight.

Edges have unit weight.

Single-source shortest path

In the MapReduce world: parallel BFS V



Problem: a lot of reading and writing to/from HDFS

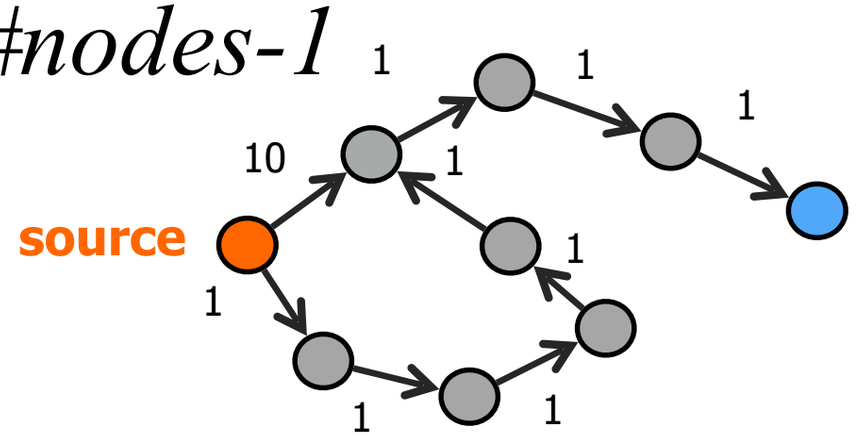
Beyond unit weight edges

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 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 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.)

- **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

The Web's graph structure

Broder et al., 1999

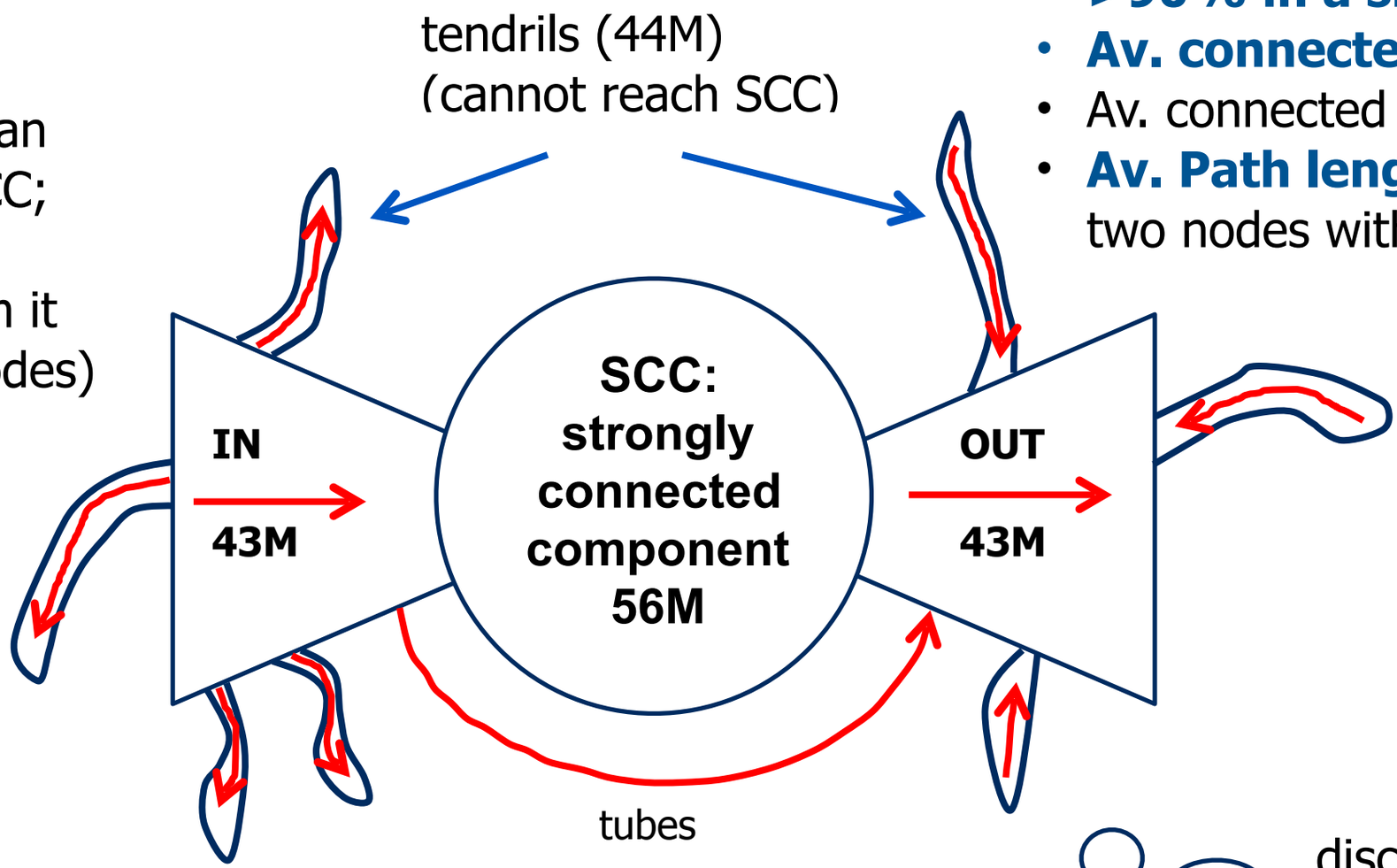
- Insights important for:
  - **Crawling** strategies
  - **Analyzing** the behaviour of algorithms that rely on link information (such as PageRank)
  - Predicting the **evolution** of Web structures
  - …

- 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



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)

disconnected components (17M)

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 $\mathbf{p_1}$ has 10 in-links and one of those is from **`yahoo.com`**, page $\mathbf{p_2}$ 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 $\mathbf{p_x}$ links to page $\mathbf{p_y}$, then the creator of $\mathbf{p_x}$ implicitly transfers some **importance** to page $\mathbf{p_y}$
  - **`yahoo.com`** is an important page, many pages point to it
    - Pages linked to from **`yahoo.com`** are also likely to be important

- Pages **distributes** "importance" through outlinks

- Simple PageRank (iteratively):

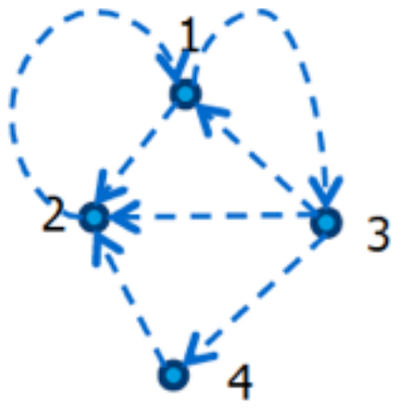
$$PageRank_{i+1}(v) = \sum_{u \to v} \frac{PageRank_i(u)}{N_u}$$

out-degree of node 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

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

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

initialize PageRank vector $\vec{R}$

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

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

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

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

PageRank vector converges eventually

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

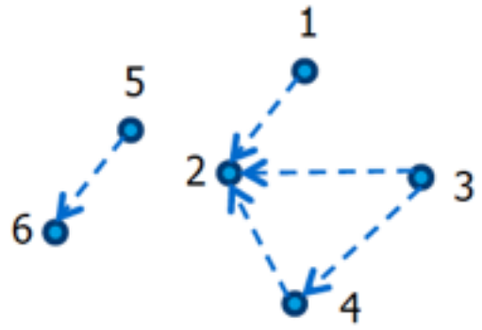
$$W^{17} \times \vec{R}' = \begin{pmatrix} 0.40 \\ 0.34 \\ 0.20 \\ 0.07 \end{pmatrix}$$

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

Reality



disconnected components

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

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

initialize PageRank vector $\vec{R}$

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

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

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

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

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

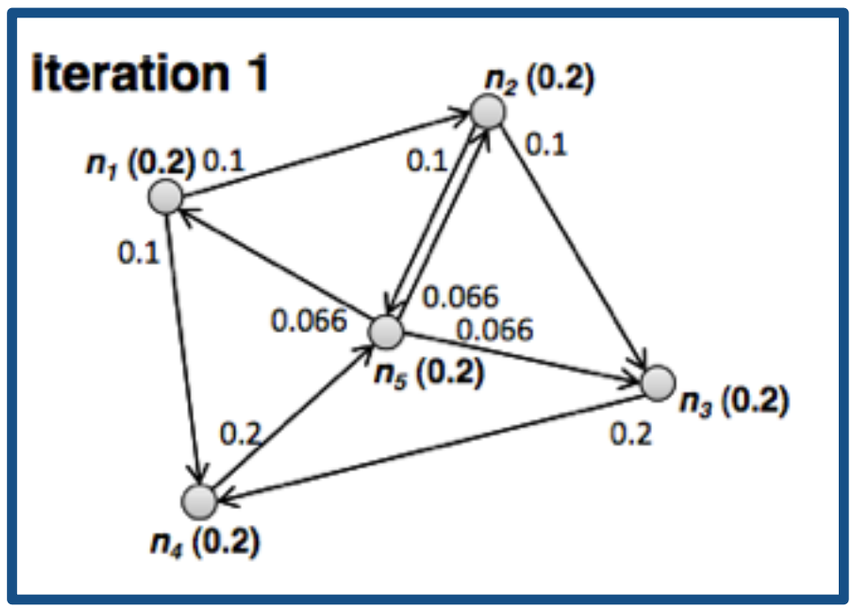
An informal sketch

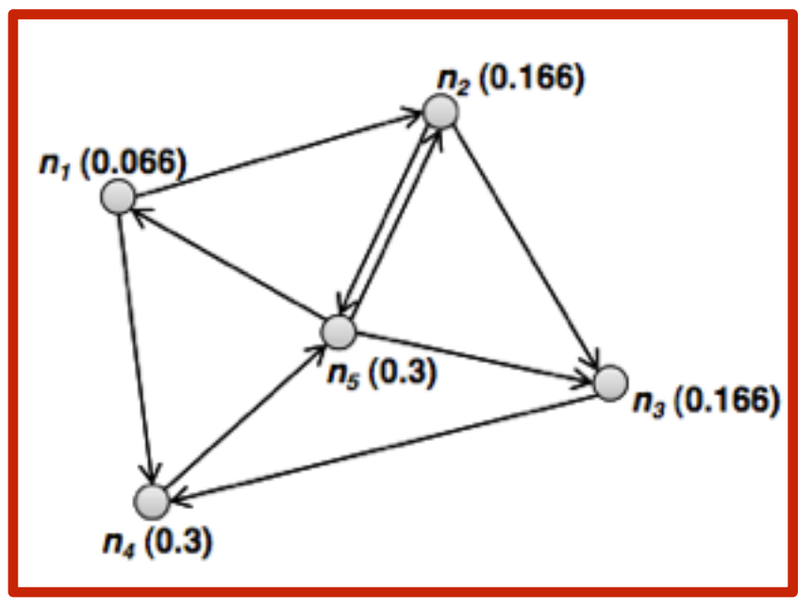
- At each iteration:
  - **[MAPPER]** a node passes its PageRank "contributions" to the nodes it is connected to
  - **[REDUCER]** each node sums up all PageRank contributions that have been passed to it and updates its PageRank score

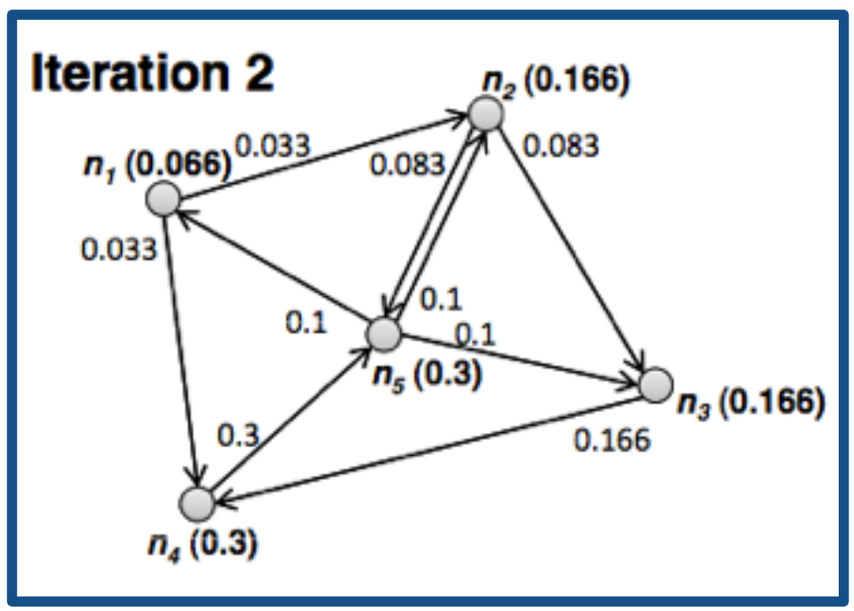
PageRank in MapReduce

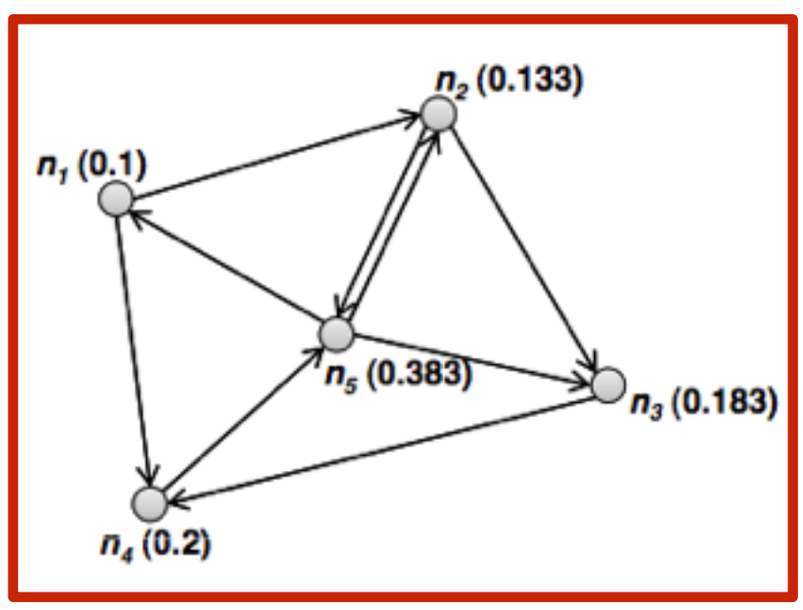
An informal sketch

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









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

PageRank in MapReduce

Pseudocode: simplified PageRank

```
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                       ▷ Sum incoming PageRank contributions
9:         M.PageRank ← s
10:        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)

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

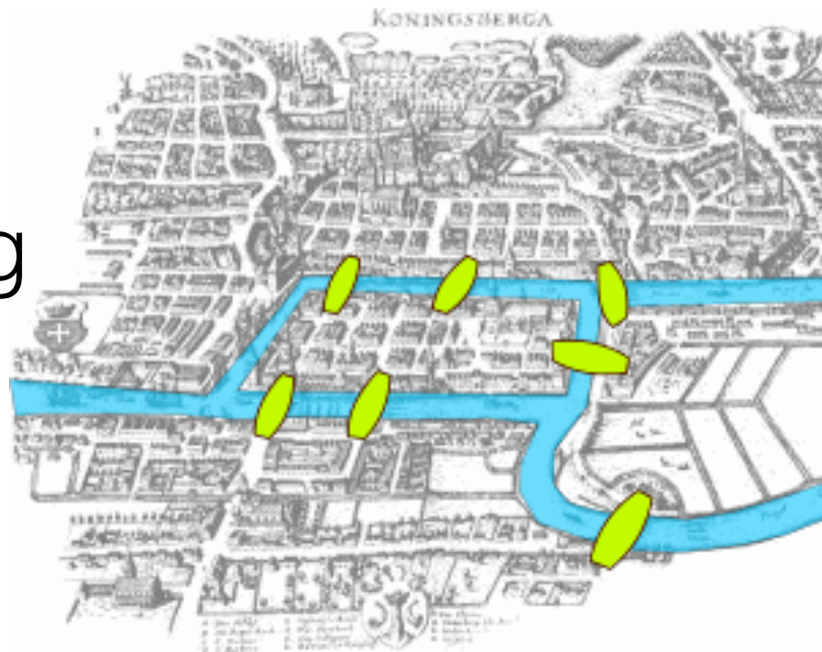
Enter Pregel (2010)

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)

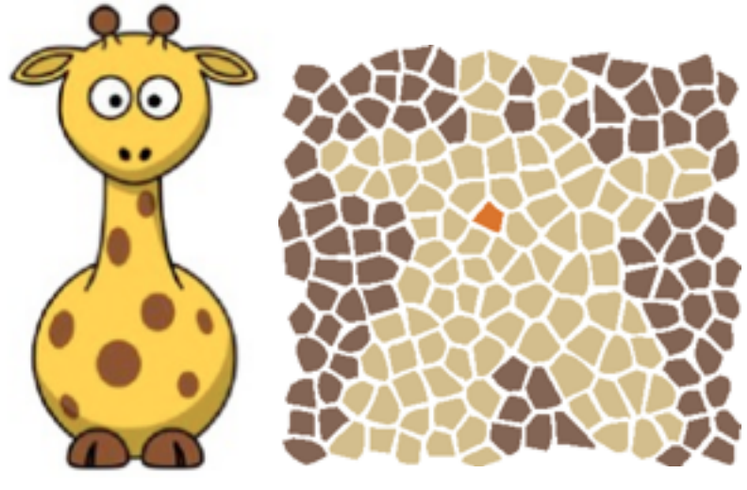


Graph processing in Hadoop

- **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

- **Disadvantage**: iterative algorithms are slow
  - Lots of reading/writing to and from disk

- **Advantage**: no additional libraries needed

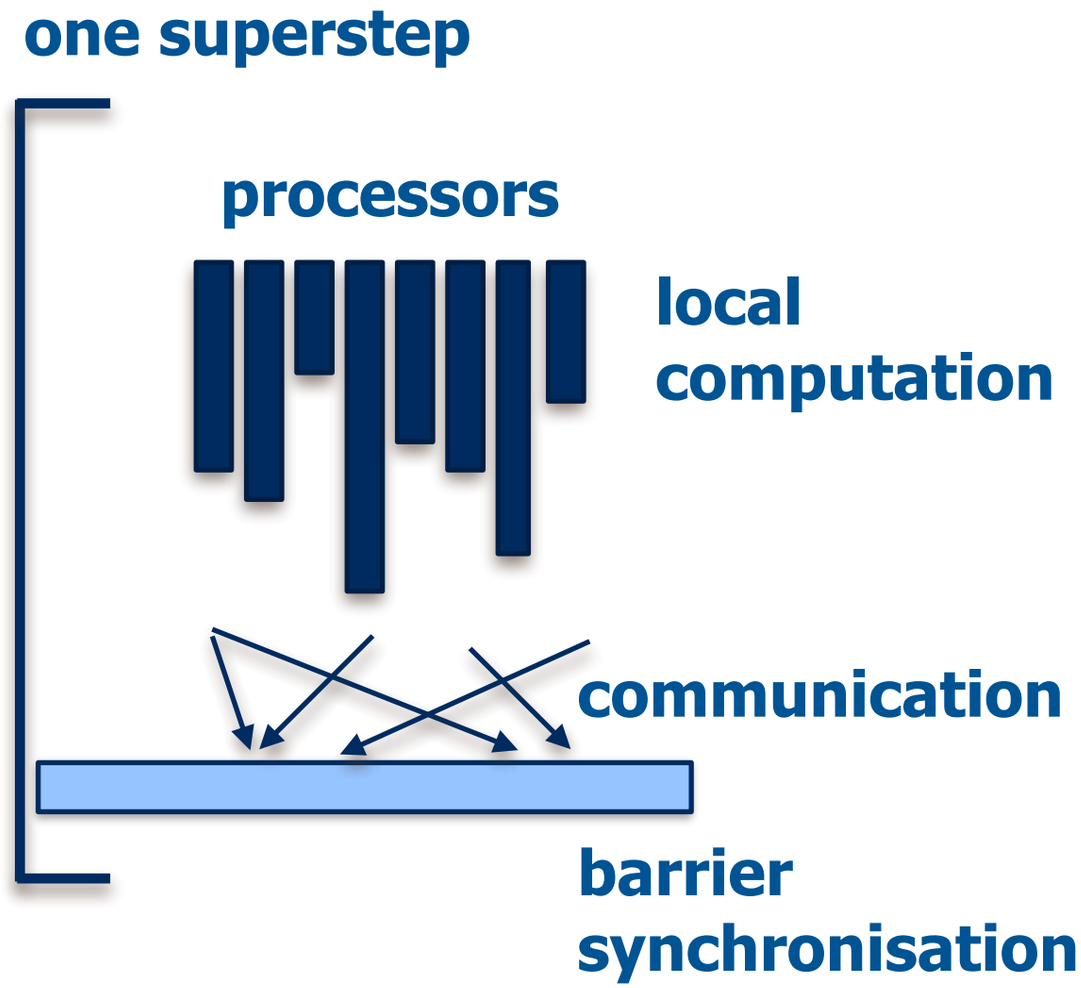
"Many distributed graph computing systems have been proposed to conduct all kinds of data processing and data analytics in massive graphs, including **Pregel**, **Giraph**, **GraphLab**, **PowerGraph**, **GraphX**, **Mizan**, **GPS**, **Giraph++**, **Pregelix**, **Pregel+**, and **Blogel**."

A bit of theory:

Bulk Synchronous Parallel or BSP

Bulk Synchronous Parallel

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



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

Bulk Synchronous Parallel

Supersteps consist of **three phases**

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)

Communication: exchange of data between processes (put and get); one-sided communication

Barrier synchronisation: all processes wait until everyone has finished the communication step

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

THE END