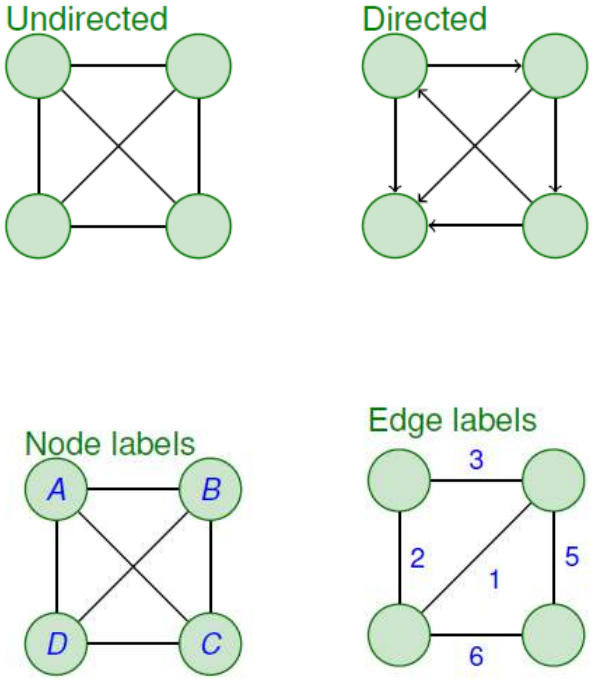
Graph algorithms in DataFrames

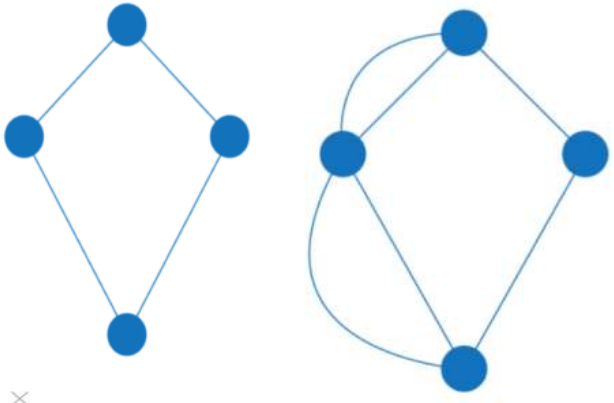
GRAPH TYPES

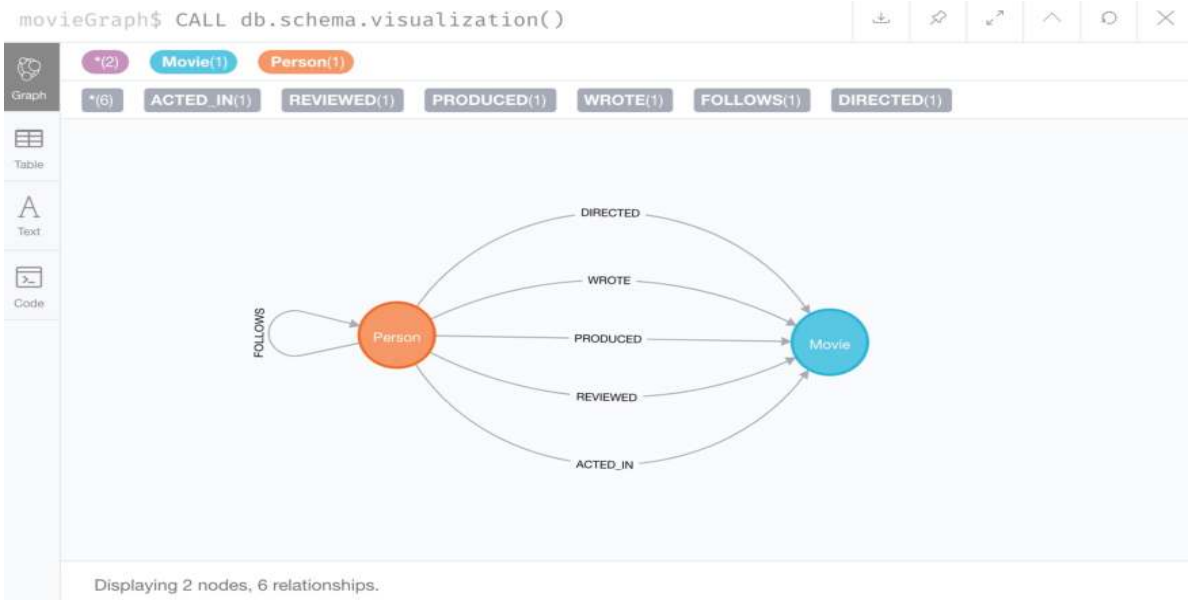
- *directed* : social network, bibliographic citations, hypertext web, semantic web, recommendation graph, evolution graph, etc.
- *undirected* : road network, collaboration network, co-occurrence graphs,
- *labeled* :
  - nodes: name, age, content
  - arcs: friendships, cost, duration, ..

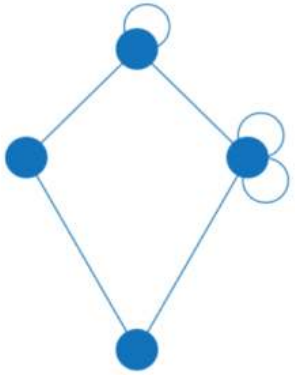


GRAPH TYPES

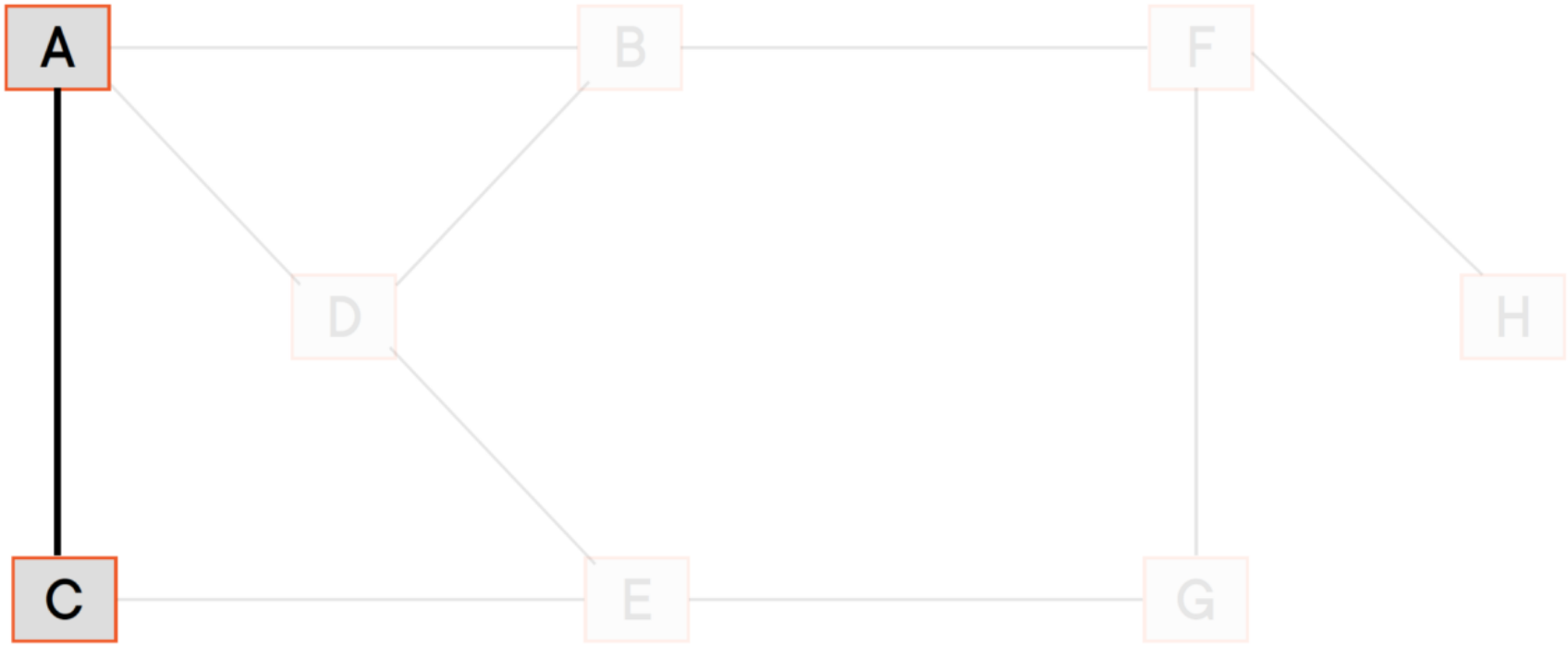
- *simple* : a single edge between each pair of nodes
- *multigraph* : several edges between each pair of nodes
- *pseudograph* : multiple edges and loops





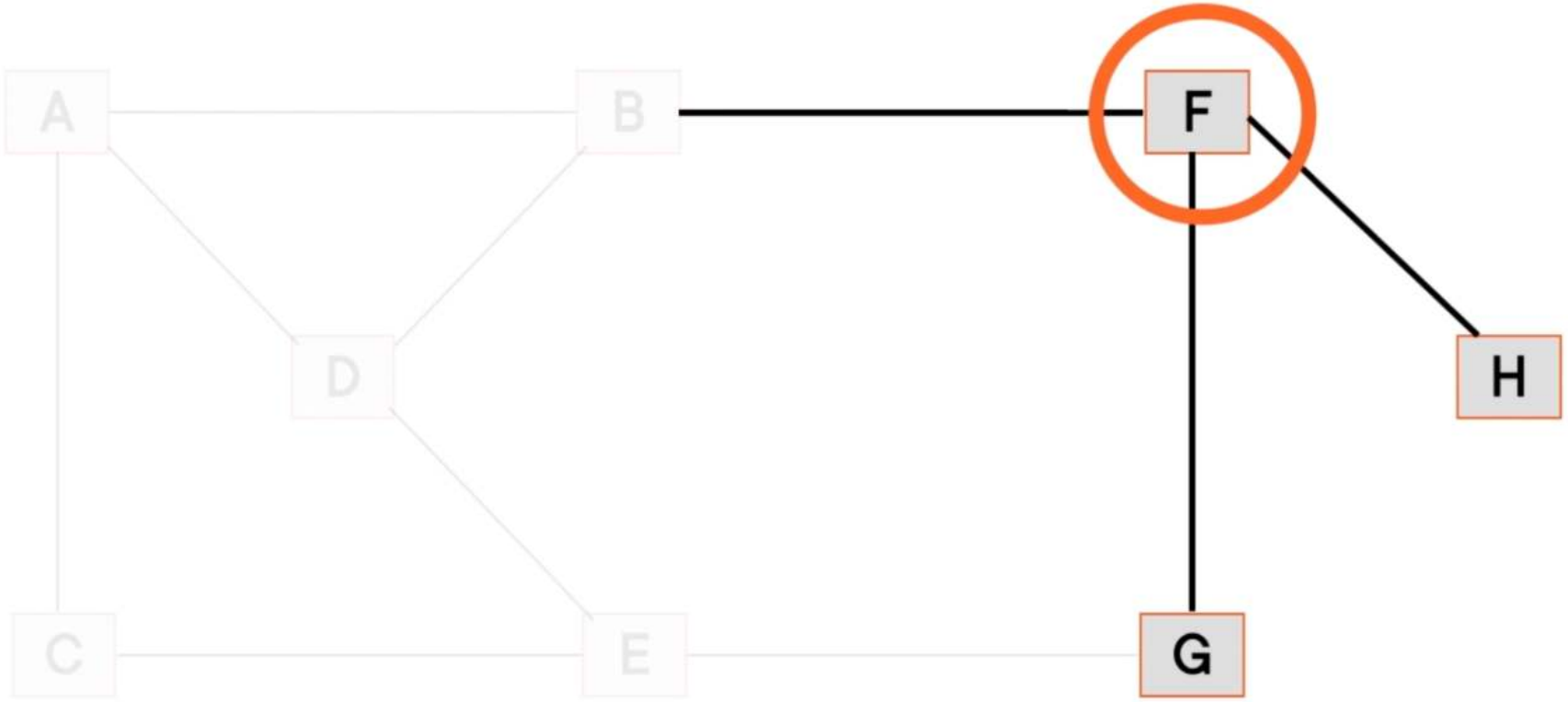


Undirected graph: Adjacent nodes



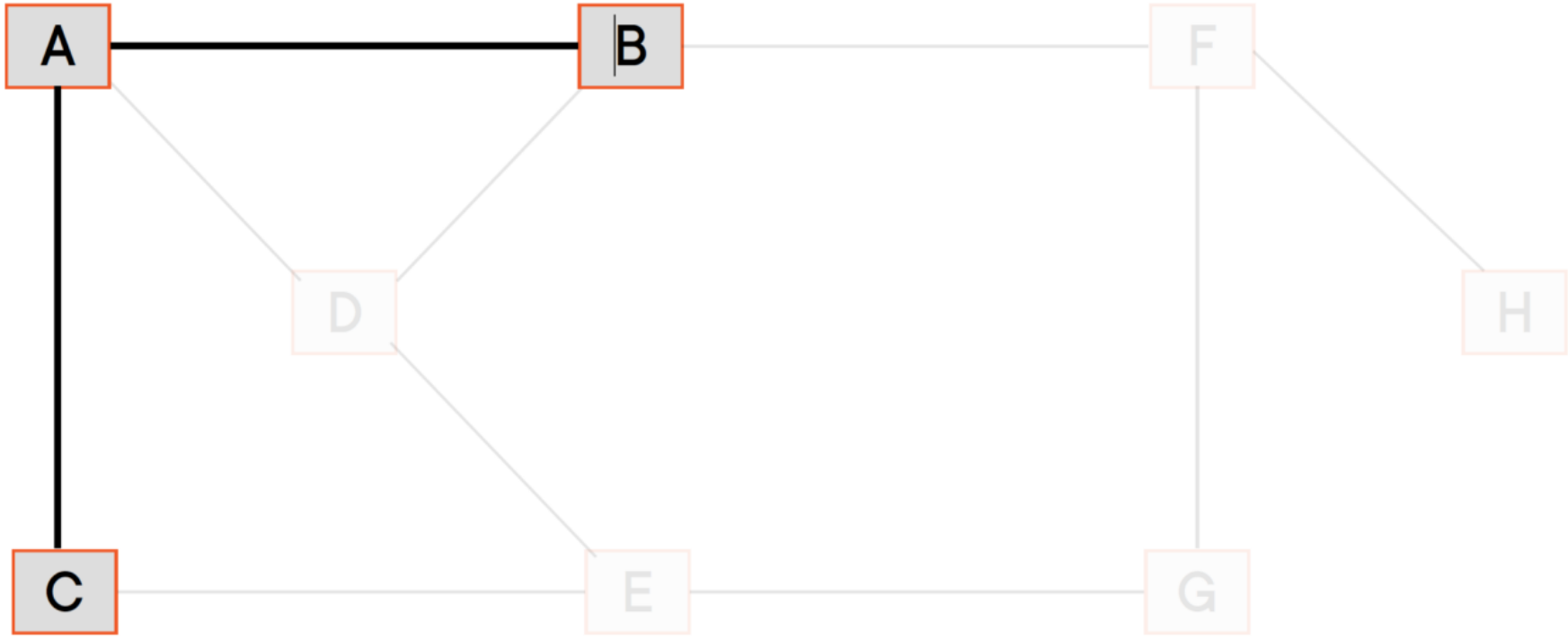
- There is an edge that connects them

Undirected graph: Degree of a node



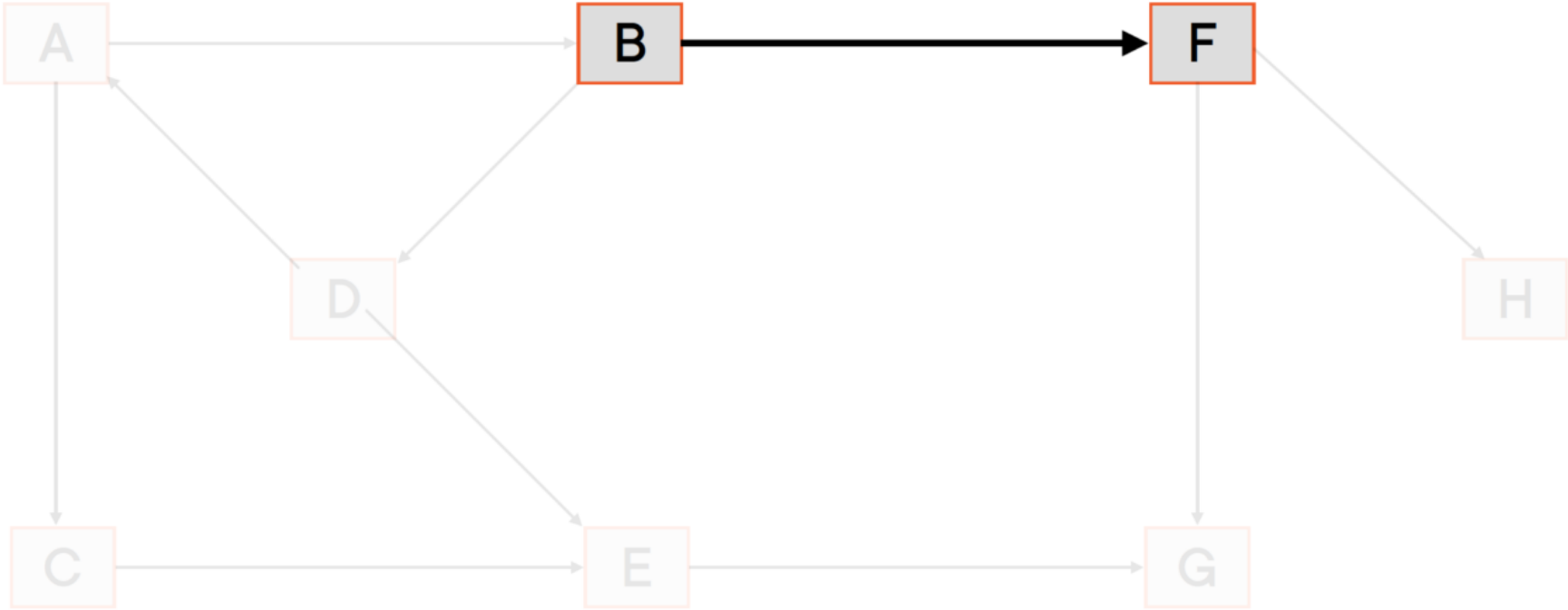
Vertex F has degree 3 (number of incident edges)

Path in an undirected graph



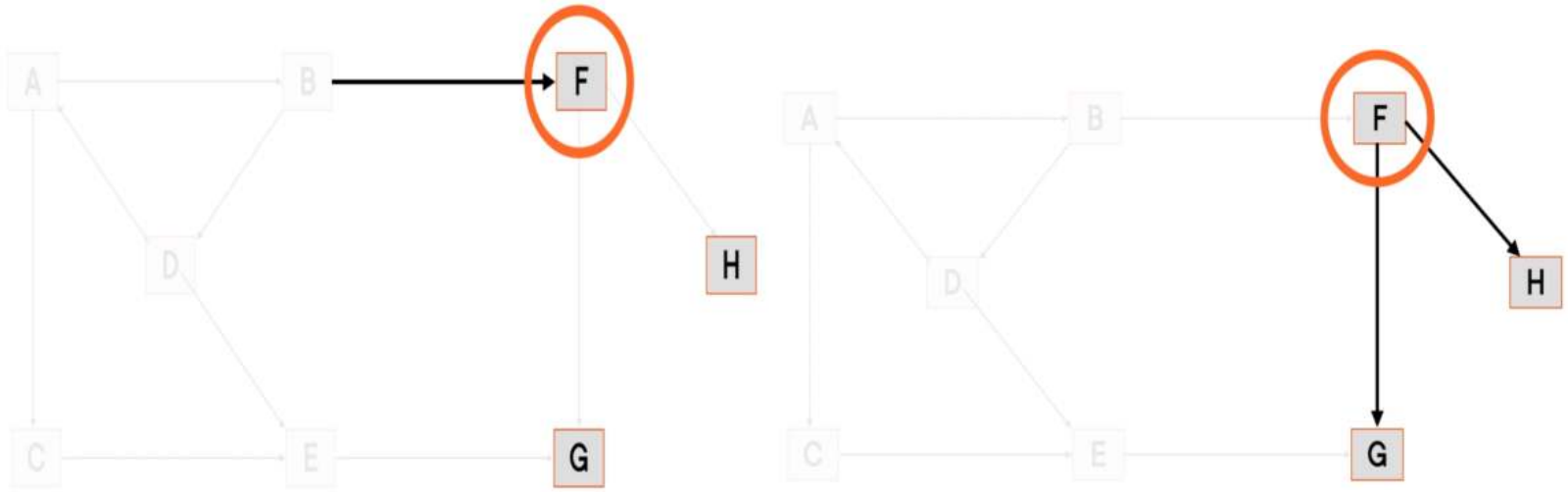
Finite sequence of consecutive edges, connecting B to C (in both directions)

Adjacent nodes in a directed graph



Node B is adjacent to node F because there is a path from B to F
(F is not adjacent to B)

Degree in/out in a directed graph

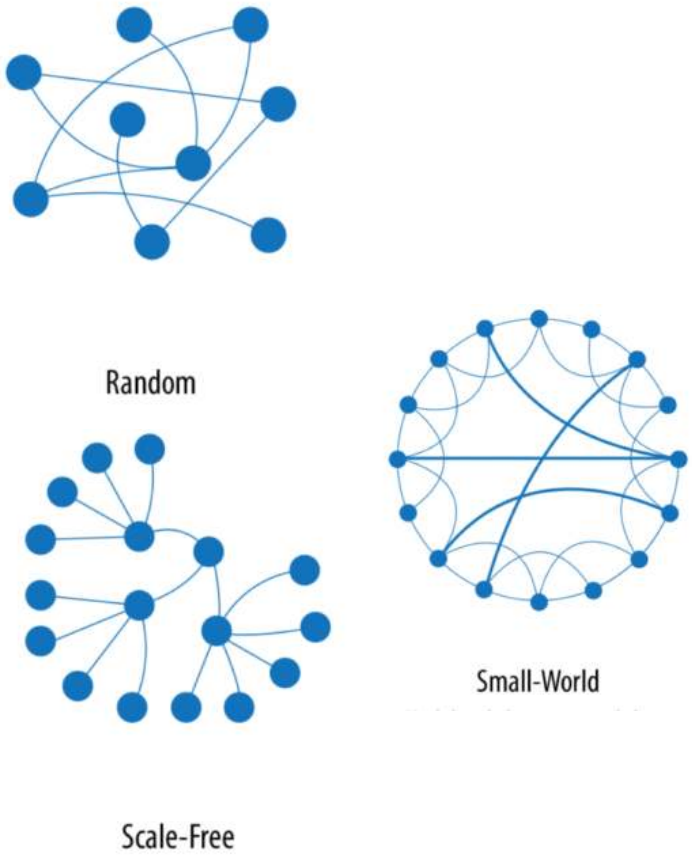


In degree of F is 1, out degree of F is 2

Types of networks

Three representative types:

- **Random**: all nodes have the same probability of being connected to another node
- **Small world:** very short paths between all nodes (e.g. social networks)
- **Scale-free**: power law for node degree distribution, some very connected nodes and many poorly connected nodes (e.g. WWW)



TYPES OF GRAPH ALGORITHMS

Exploration/ Pathfinding

Find optimal paths or assess availability and quality routes

Centrality/ Importance

Find the importance of different nodes of the network

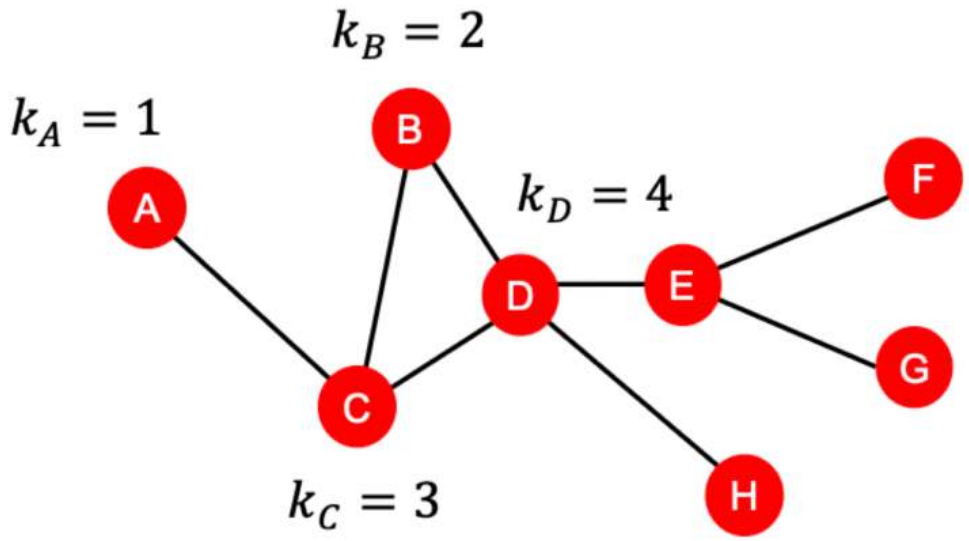
Detection of communities

Find clusters or partitions

CENTRALITY-BASED ALGORITHMS

Degree centrality

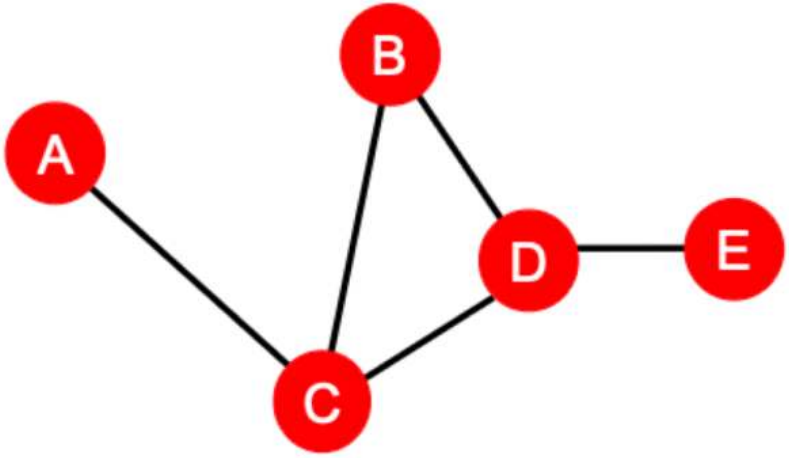
- The simplest among centrality-based algorithms
- **Degree**: Number of edges of the node (number of neighbors)
- **Application**: find popular nodes (eg popular people in a social network)
- All nodes are treated the same → does not take into account the importance of nodes



Closeness Centrality

- Important nodes have the shortest distances to all other nodes
- Measures the average farness (inverse distance) from a node to all other nodes.
- **Application**: detect nodes capable of spreading information in a subgraph (eg people in good position to control and acquire information and resources within an organization)

$$c_v = \frac{1}{\sum_{u \neq v} d(u, v)}$$



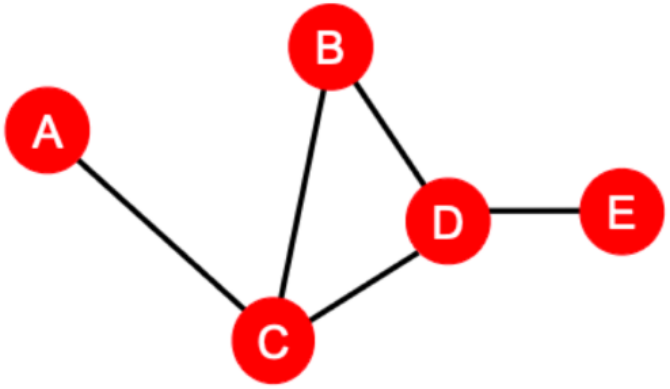
$c_A = 1/(2 + 1 + 2 + 3) = 1/8$
(A-C-B, A-C, A-C-D, A-C-D-E)

$c_D = 1/(2 + 1 + 1 + 1) = 1/5$
(D-C-A, D-B, D-C, D-E)

Betweenness centrality

- Important nodes are on many of the shortest paths between other nodes (central nodes for the information circulating in the graph)
- Captures a node's role in allowing information to pass from one part of the network to the other
- **Application**: detect the degree of influence of a node on the flow of information in a graph (e.g a person capable of connecting people from different companies, passing ideas between communities, etc.)

$$c_v = \sum_{s \neq v \neq t} \frac{d(s,v,t)}{d(s,t)}$$



$$c_A = c_B = c_E = 0$$

$$c_C = 3$$

(A-C-B, A-C-D, A-C-D-E)

$$c_D = 3$$

(A-C-D-E, B-D-E, C-D-E)

Eigenvector centrality

- scores relative to all nodes in the network
- scores influenced more by connections to high-scoring nodes than by connections to low-scoring nodes.
- Important nodes: surrounded by important neighbors $u \in N(v)$
- The centrality score of a node is the sum of the centrality scores of its neighbors (recursive definition)

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u$$

λ is a constant used for normalization (the largest eigenvalue of the adjacency matrix)

PageRank: Importance of Web Pages

Eigenvector centrality

Rewriting in matrix form:

$$c_v = \frac{1}{\lambda}\sum_{u \in N(v)} c_u \quad \Longleftrightarrow \quad \lambda c = Ac$$

A = Adjacency matrix (A_{uv} = 1 si u ∈ N(v))

c_{max} = eigenvector of the largest eigenvalue λ_{max}

- variants: PageRank and Katz centrality score

PageRank: principles

Hyperlinks = recommendations

Principle:

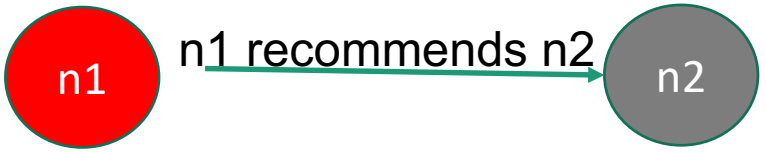
- Pages with a lot of recommendations are more important
- Also important: *who* gives the recommendation

- Principles:
  - *being recommended by Yahoo! is better than by X*
  - *the recommendation counts less if Yahoo! recommends lots of pages*

→ importance of a page depends on the **numbe**r and on the **quality** (importance) of the one who recommends it

Simplified PageRank

Recommendation given by **n1** to **n2** :



$$PR(n1) * \frac{1}{|out(n1)|}$$

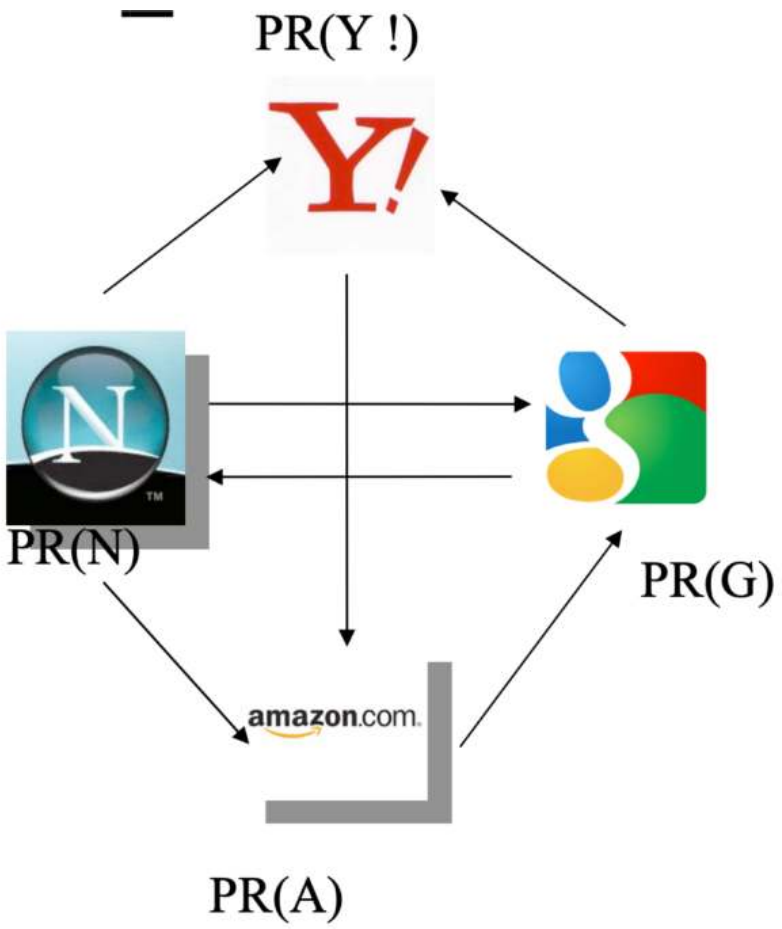
PR(n1) = importance of n1
|out(n1)| = out degree of n1

The importance of **n2** is the sum of the recommendations received:

$$PR(n2) = \sum_i PR(ni) * \frac{1}{|out(ni)|}$$

ni = pages that recommend n2

Example



PR(A) = PR(N) /3 + PR(Y)
PR(Y) = PR(N) /3 + PR(G)/2
PR(N) = PR(G)/2
PR(G) = PR(A) + PR(N)/3

Computation of PR values

Linear system resolution

- 4 equations with 4 unknowns
- test the existence and uniqueness of the solution

$$PR(A) = PR(N)/3 + PR(Y)$$
$$PR(Y) = PR(N)/3 + PR(G)/2$$
$$PR(N) = PR(G)/2$$
$$PR(G) = PR(A) + PR(N)/3$$

→ add the constraint $PR(A)+PR(Y)+PR(N)+PR(G) = 1$ to ensure uniqueness

Observation:

- large linear system, many pages without outgoing links

⇒ direct calculation methods (e.g. Gauss method) are more expensive than *iterative* methods

Matrix representation

We consider *n* pages, for each page i, we note:

- *out(j)* : the set of pages i referenced by j

- *M(wij)* : the adjacency matrix associated with the Web graph
  - *wij* : fraction of the importance of j which is given to i (*wij = 1/|out(j)|* , if j has outgoing links, *wij =0* otherwise)

⇒ line i = importance fractions received by i
column j = distribution of importance of j

(for pages j with outgoing links, *the sum of the elements on the columns* is 1)

- *Pri* : importance of page i
- *PR(PR1, PR2,..., PRn)* : vector of unknowns (importance)

Iterative computation algorithm

- **Input**: a graph with n nodes
- **Output**: the PageRank vector
- **Initialization**: $PR^0 = [1/n, ..., 1/n]$
- Recompute $PR^{(k)}$ at each iteration k:

$$PR^{(k)} = M * PR^{(k-1)}$$

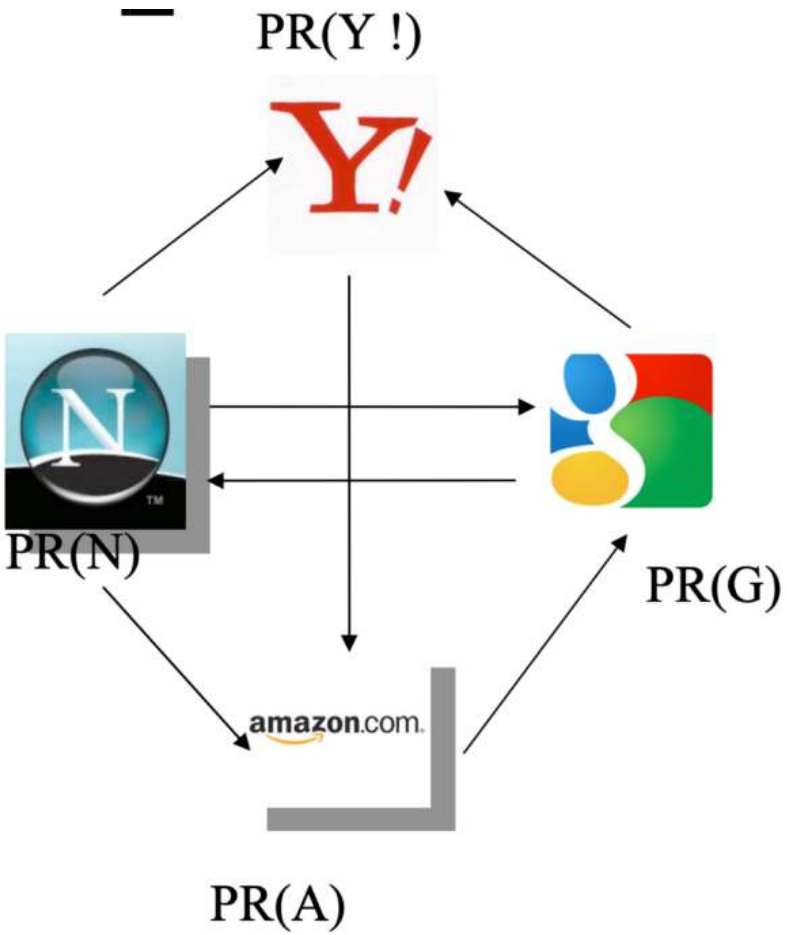
- Stop the computation (convergence) :

$$\frac{\sum |PR_i^k - PR_i^{(k-1)}|}{\| PR^k \|_1} < \varepsilon, \varepsilon \in (0,1)$$

- The PR vector obtained at convergence satisfies the condition:

$$PR = M * PR$$

Example

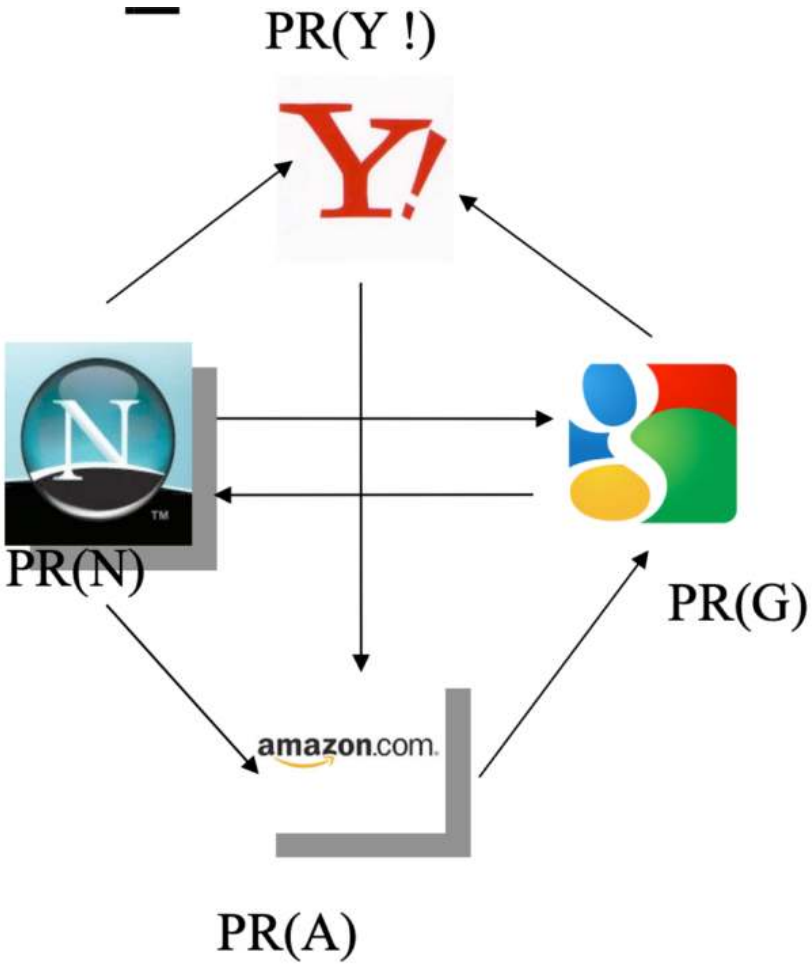


PR(A) = PR(N) /3 + PR(Y)
PR(Y) = PR(N) /3 + PR(G)/2
PR(N) = PR(G)/2
PR(G) = PR(A) + PR(N)/3

W_{YN}=1/3

	Y	N	A	G
Y		1/3		1/2
N				1/2
A	1	1/3		
G		1/3	1	

Example

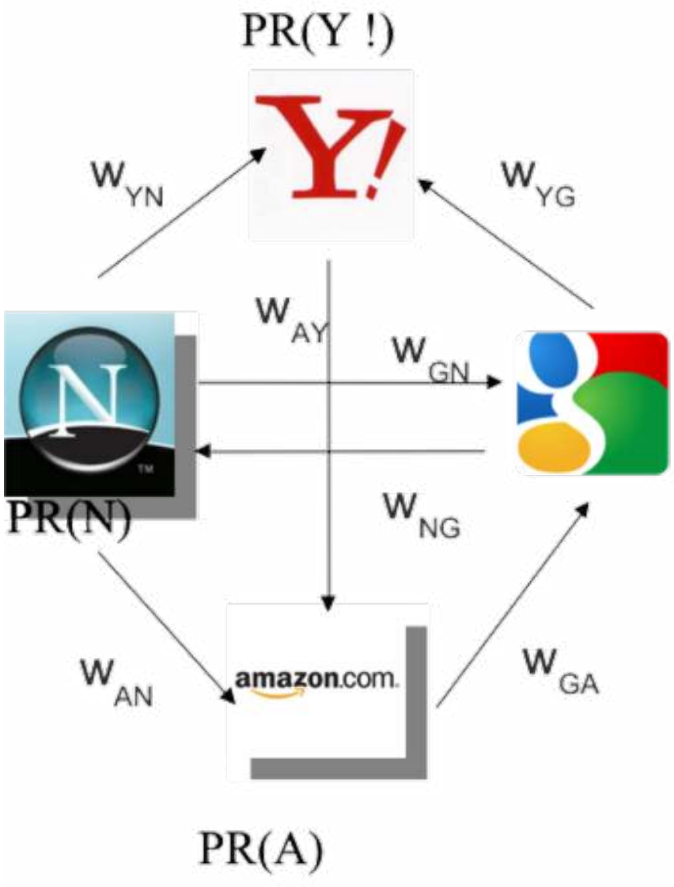


PR(A) = PR(N) /3 + PR(Y)
PR(Y) = PR(N) /3 + PR(G)/2
PR(N) = PR(G)/2
PR(G) = PR(A) + PR(N)/3

		Y	N	A	G		
PR(Y)	Y		1/3		1/2		PR(Y)
PR(N)	N				1/2	*	PR(N)
PR(A)	A	1	1/3				PR(A)
PR(G)	G		1/3	1			PR(G)

PR^k = M * $PR^{(k-1)}$

Example: computation at each iteration



$$PR(A) = PR(N) * w_{AN} + PR(Y) * w_{AY}$$

$$PR(Y) = PR(N) * w_{YN} + PR(G) * w_{YG}$$

$$PR(N) = PR(G) * w_{NG}$$

$$PR(G) = PR(A) * w_{GA} + PR(N) * w_{GN}$$

$$PR^{K} = M * PR^{(K-1)}$$

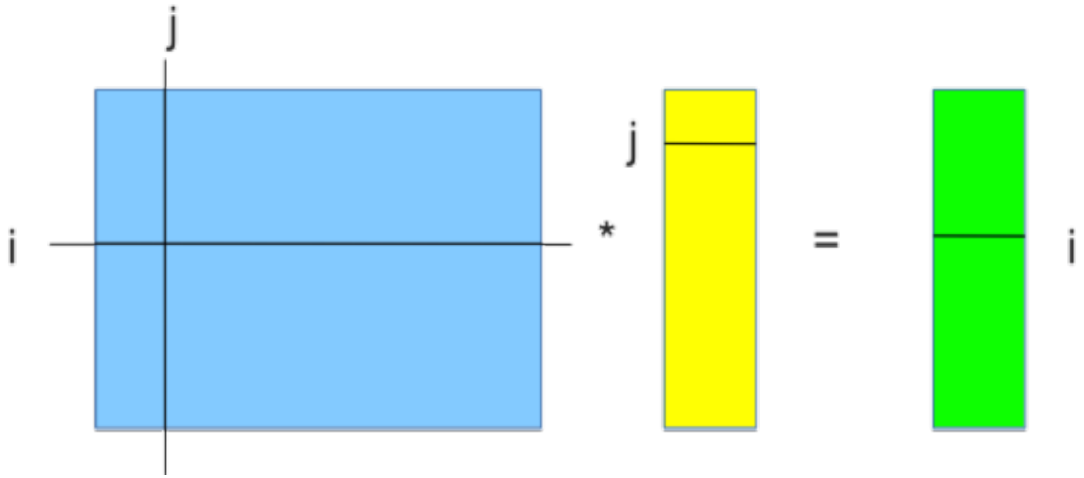
	Y	N	A	G
PR(Y)		w_{YN}		w_{YG}
PR(N)				w_{NG}
PR(A)	w_{AY}	w_{AN}		
PR(G)		w_{GN}	w_{GA}	

Example: update for each page

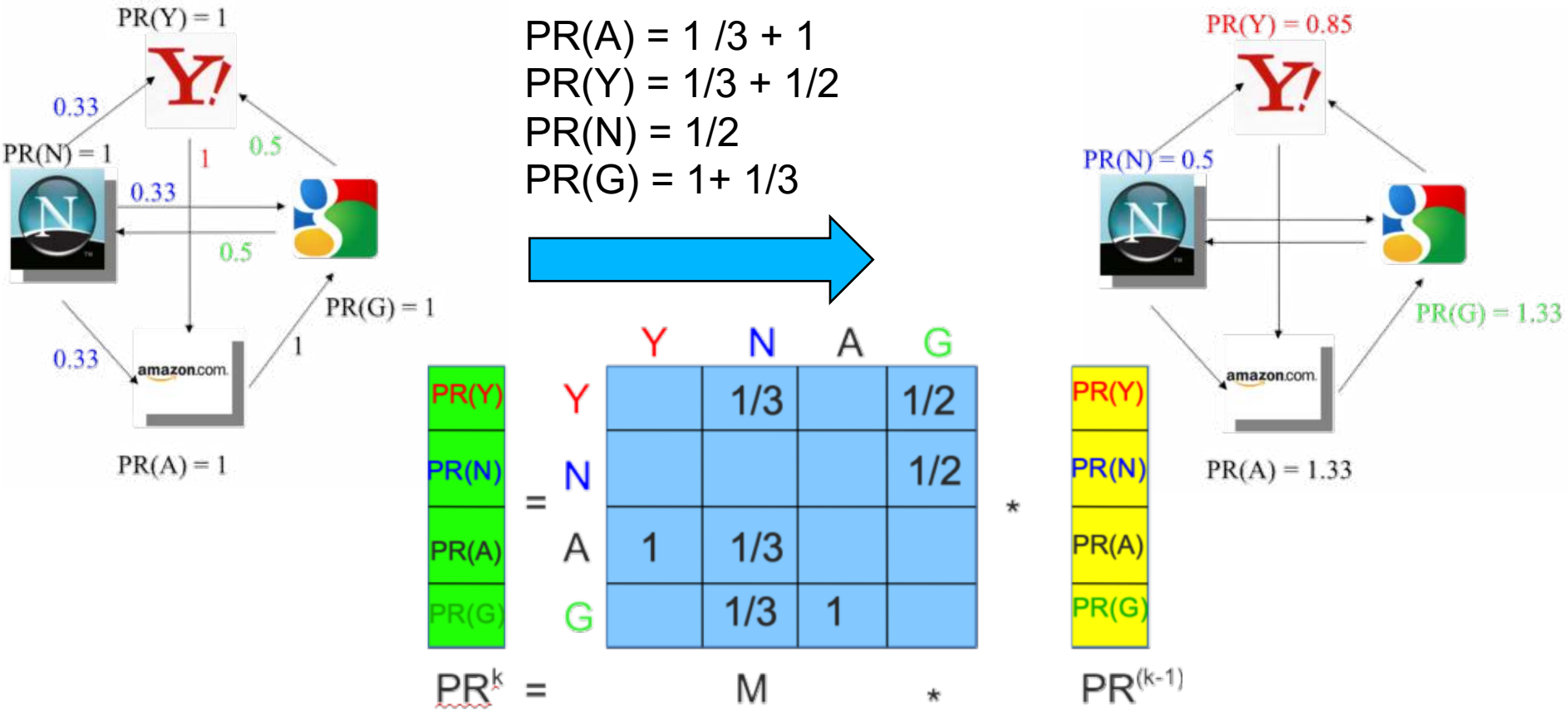
Update of Pr_i :

$$PR_i = \sum_j w_{ij} * PR_j$$

$$PR_i = \sum_j \frac{1}{|out(j)|} * PR_j$$

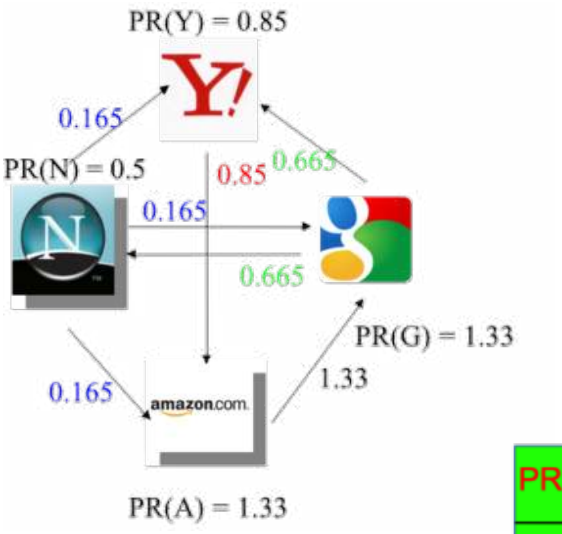


Example: first iteration

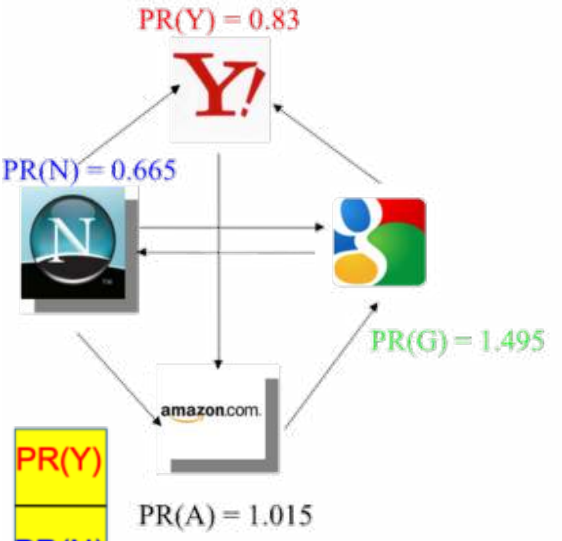


Weight of the arcs for the example: importance fraction ($w_{ij} * PR_j$)

Example: second iteration



PR(A) = 0.5 /3 + 0.85
PR(Y) = 0.5/3 + 1.33/2
PR(N) = 1.33/2
PR(G) = 1.33+ 0.5/3



		Y	N	A	G		
PR(Y)	Y		1/3		1/2	*	PR(Y)
PR(N)	N				1/2		PR(N)
PR(A)	A	1	1/3				PR(A)
PR(G)	G		1/3	1			PR(G)

$PR^k = M * PR^{(k-1)}$

Weight of the arcs for the example: importance fraction ($w_{ij} * PR_j$)

Complete iterative algorithm

- **Input**: a graph with n nodes
- **Output**: the PageRank vector
- **Initialization**: $PR^0 = [1/n, ..., 1/n]$
- Vector added at each iteration: $V = [1/n, ..., 1/n]$
- Recompute $PR^{(k)}$ at each iteration k:

$$PR^{(k)} = d*M * PR^{(k-1)} + (1-d)*V$$

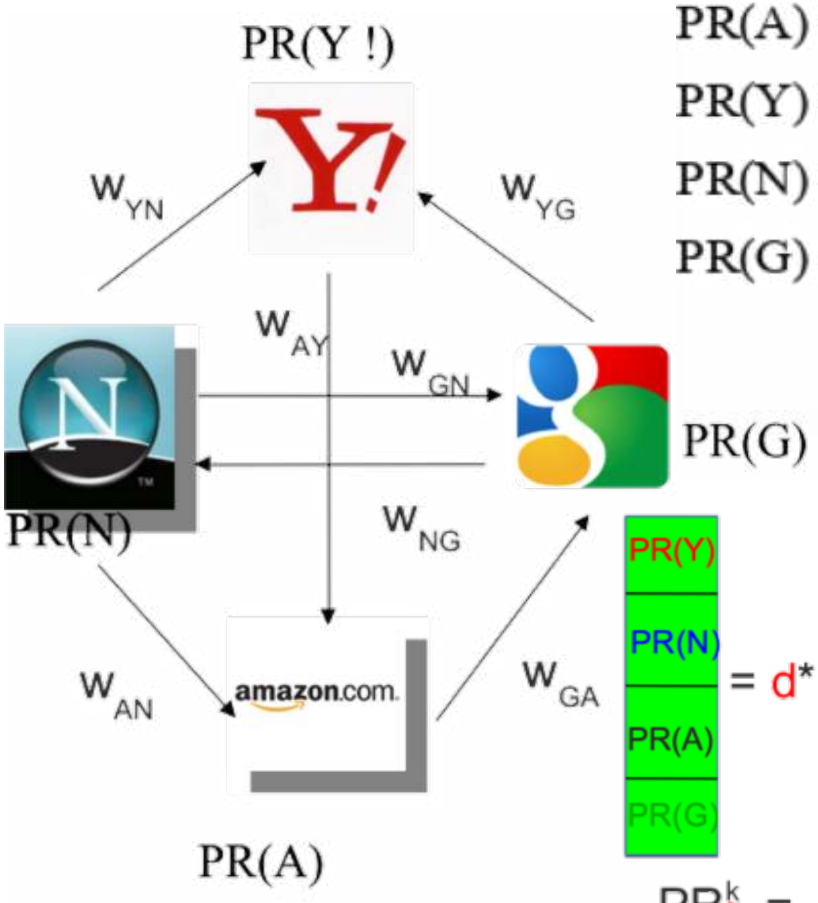
or its equivalent: $\forall i : PR_i^k = d * \sum_j PR_j^{k-1} * w_{ij} + (1-d) * v_i \quad)$

stops when: $\frac{\sum |PR_i^k - PR_i^{k-1}|}{\|PR^k\|_1} < \varepsilon, \varepsilon \in (0,1)$

- The decay factor d (usually 0.85) is used to ensure the uniqueness of the PR vector and the convergence of the iterative computation
- **Personalization** (specific importance for a set E of nodes)

$V = [v_1, ..., v_n]$ ($v_i = 1/E$ if $i \in E$, $v_i = 0$ otherwise)

Example: Complete iterative algorithm

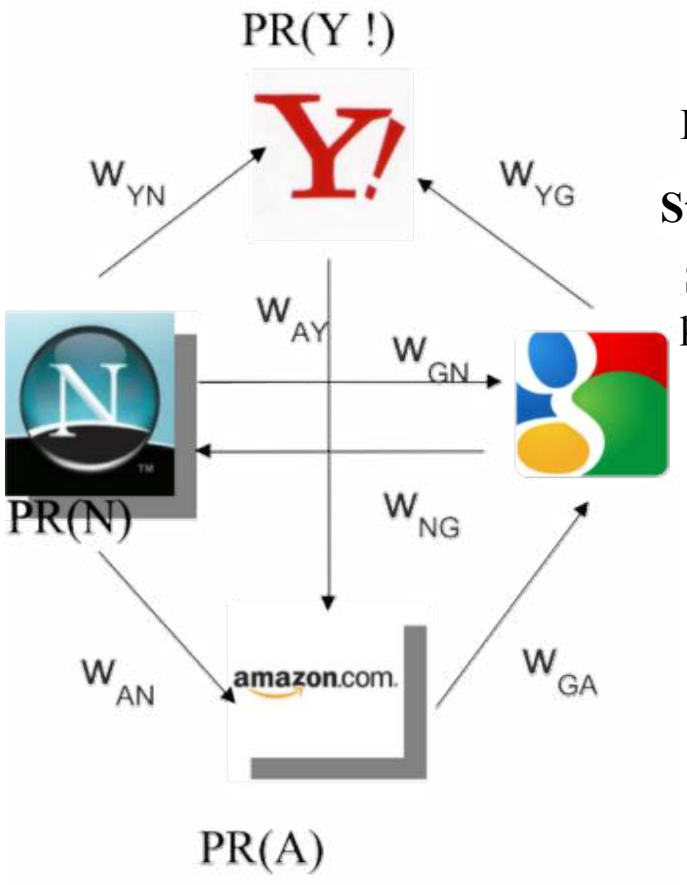


$$PR(A) = d*(PR(N)*w_{AN} + PR(Y)*w_{AY}) + (1-d)*v_A$$
$$PR(Y) = d*(PR(N)*w_{YN} + PR(G)*w_{YG}) + (1-d)*v_Y$$
$$PR(N) = d*(PR(G)*w_{NG}) + (1-d)*v_N$$
$$PR(G) = d*(PR(A)*w_{GA} + PR(N)*w_{GN}) + (1-d)*v_G$$

$$\begin{bmatrix} PR(Y) \\ PR(N) \\ PR(A) \\ PR(G) \end{bmatrix} = d* \begin{array}{c} \\ Y \\ N \\ A \\ G \end{array} \begin{array}{c} \begin{array}{cccc} Y & N & A & G \end{array} \\ \begin{bmatrix} & w_{YN} & & w_{YG} \\ & & & w_{NG} \\ w_{AY} & w_{AN} & & \\ & w_{GN} & w_{GA} & \end{bmatrix} \end{array} * \begin{bmatrix} PR(Y) \\ PR(N) \\ PR(A) \\ PR(G) \end{bmatrix} + (1-d)* \begin{bmatrix} v_Y \\ P_N \\ P_A \\ P_G \end{bmatrix}$$

$$PR^k = d* \quad M \quad * \quad PR^{(k-1)} + (1-d)* \quad V$$

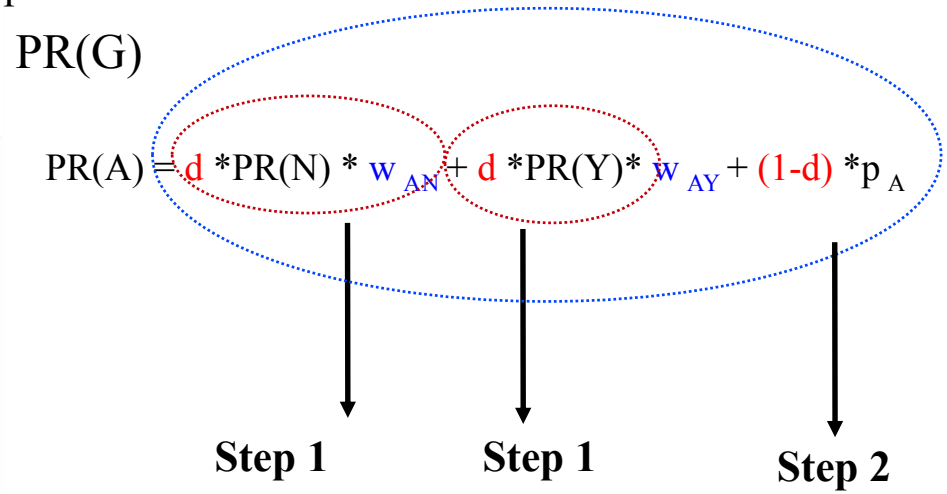
PageRank in DataFrames



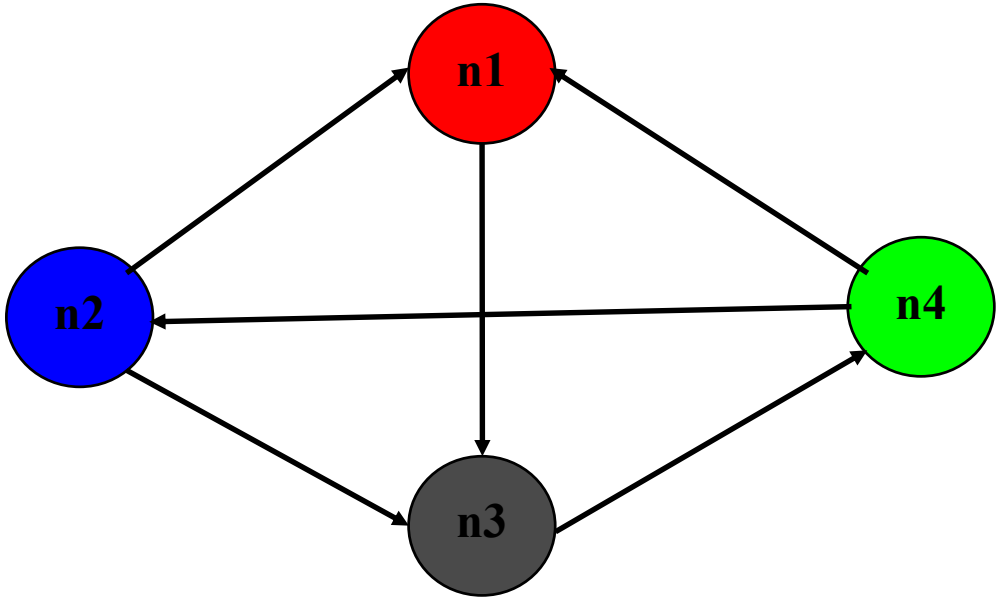
Breakdown of the calculation at each iteration **k** into two steps:

Step 1 : calculation of values $d*PR_j^k * wij$

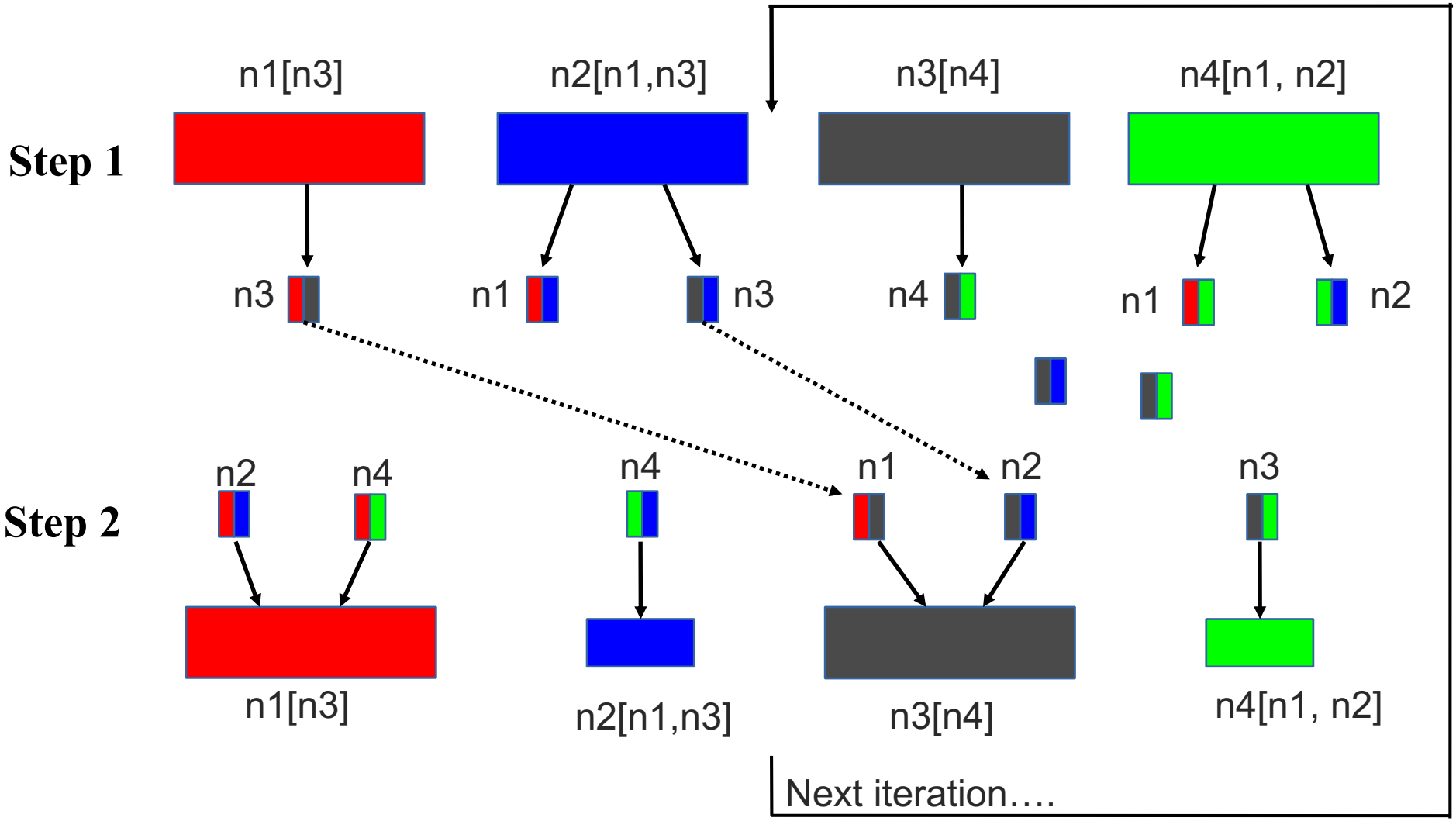
Step 2 : Sum of previous values and addition of personalization values



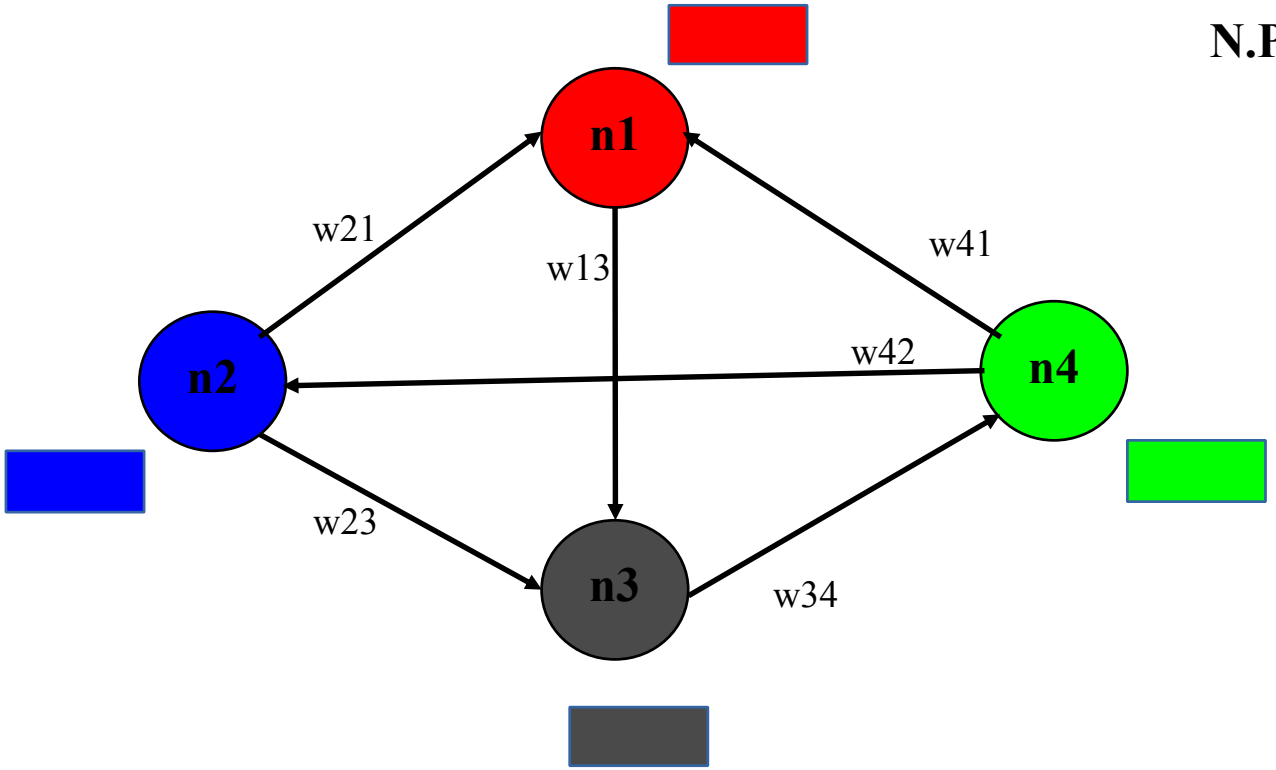
Example



PageRank in DataFrames



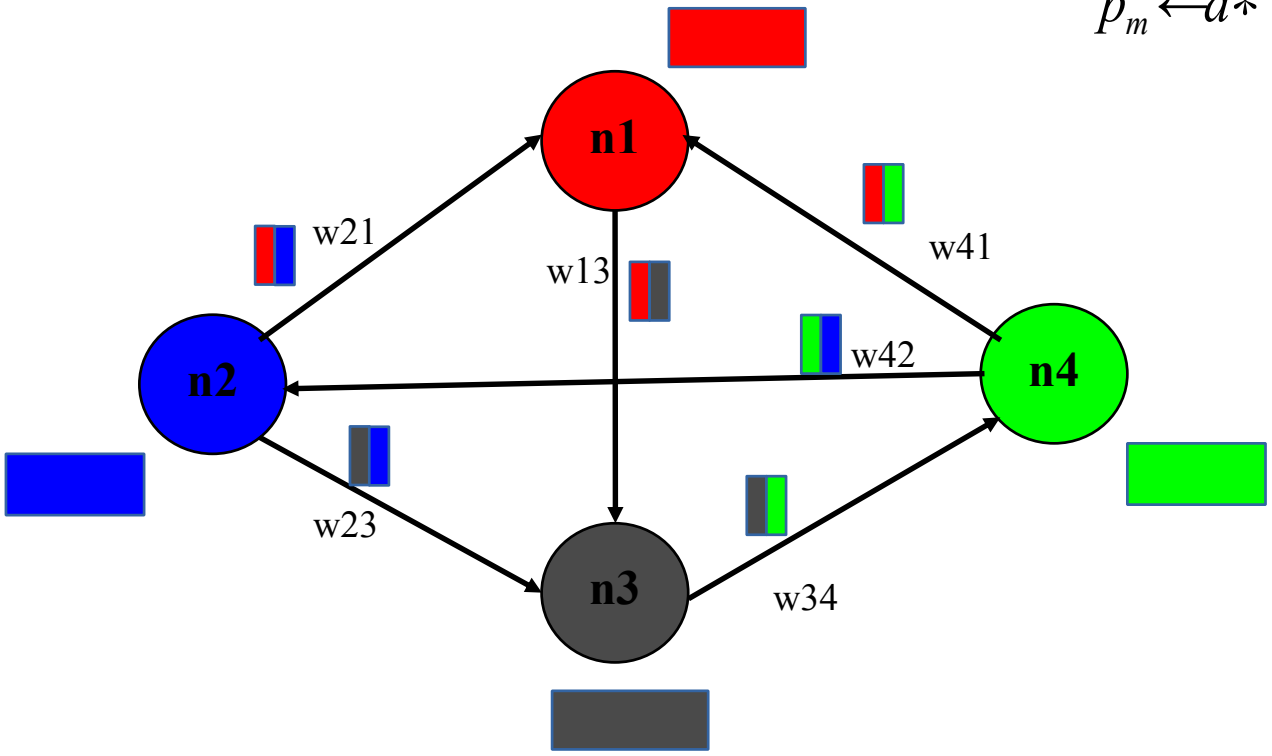
Example



N.PAGERANK

Example: **Step 1**

$$p_m \leftarrow d * w_{nm} * N.PAGERANK$$



= d * w13 *

= d * w41 *

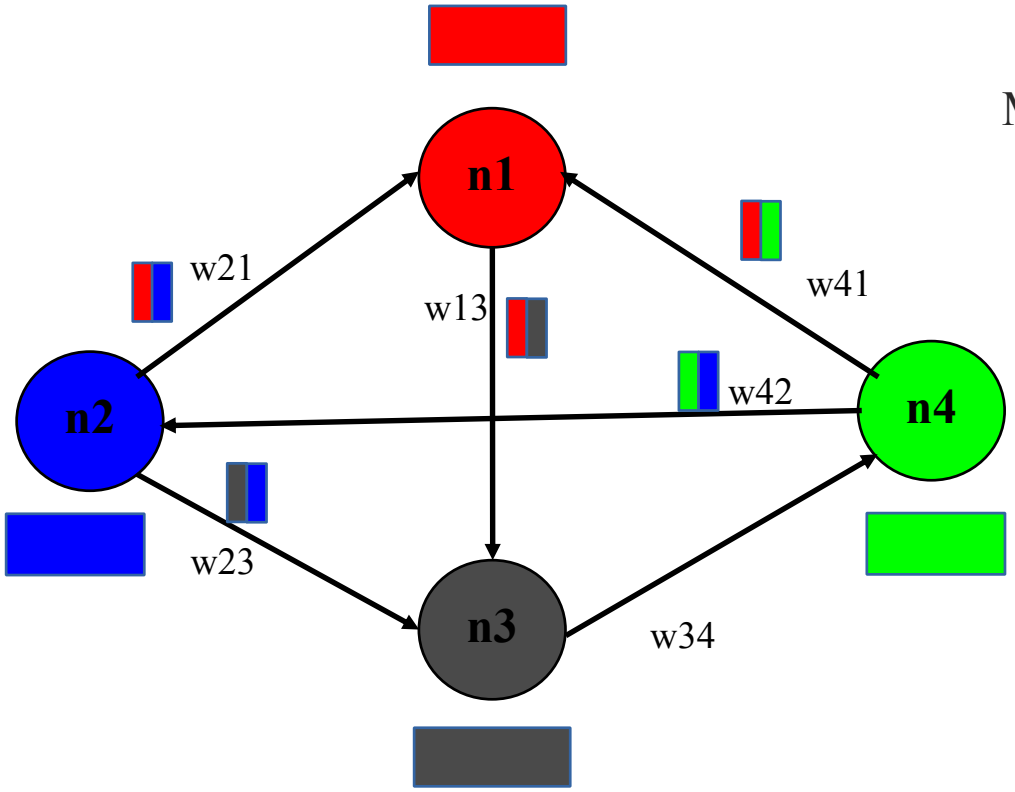
= d * w42 *

= d * w34 *

= d * w21 *

= d * w23 *

Example: **Step 2**



M.PAGERANK ← s+(1-d)*MP

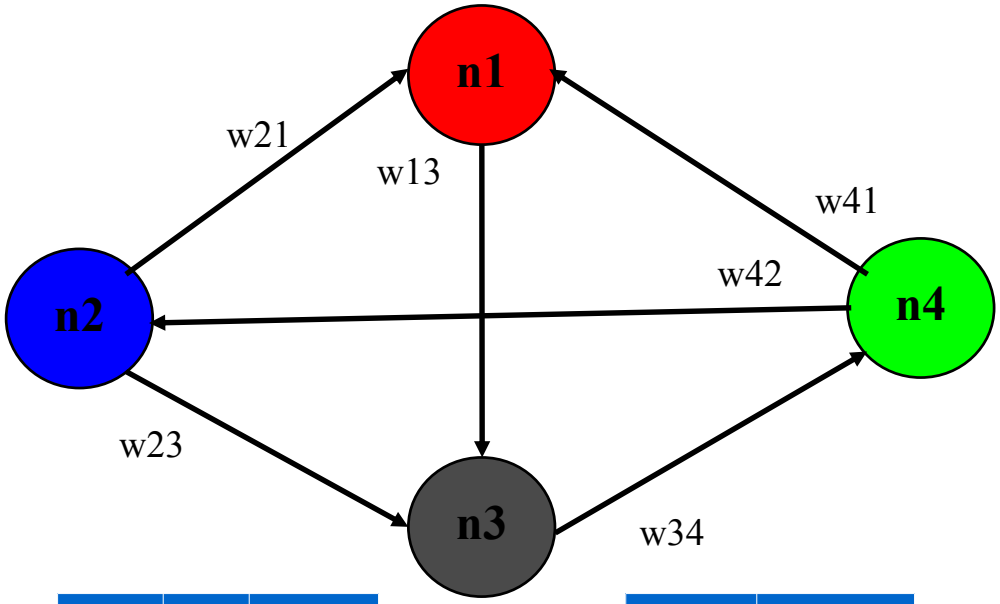
= (+) + (1-d) * n1.PERS

= ( ) + (1-d) * n2.PERS

= ( ) + (1-d) * n4.PERS

= (+) + (1-d) * n1.PERS

Example: personalized PR



d = 0.85

PR = importance to compute

P = personalization vector (wrt. n2)

S	D	w
1	3	w13
2	1	w21
2	3	w23
3	4	w34
4	1	w41
4	2	w42

graph

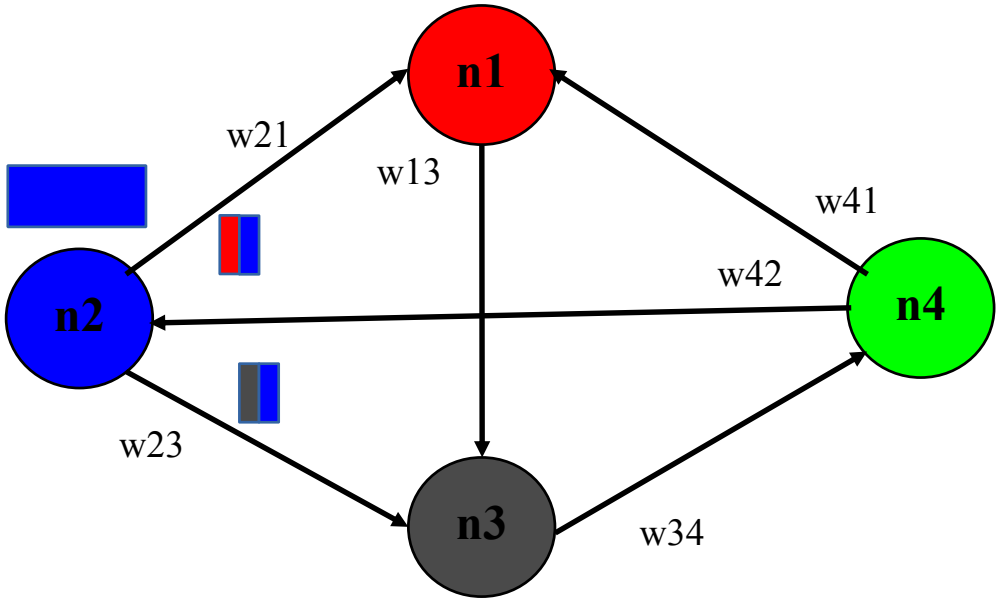
id	p
2	1

P

id	i
1	i1
2	i2
3	i3
4	i4

PR

Example: First iteration



= i2_0*d*w21

= i2_0*d*w23

S	D	w
1	3	w13
2	1	w21
2	3	w23
3	4	w34
4	1	w41
4	2	w42

graph

id	i
2	i2_0

= i2_0=1

PR=P

First iteration

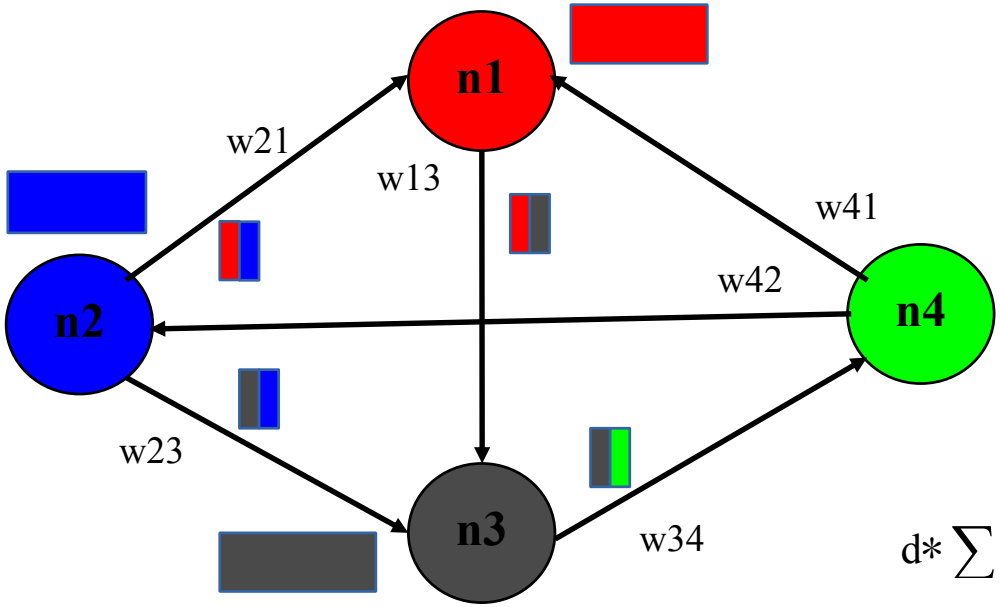
D	next-i
1	i2_0 * d *w21
3	i2_0 * d *w23

$d* \sum i_j^0 * w_{ji}$

id	i
1	i2_0 * d * w21
3	i2_0 * d * w23
2	(1-d) * 1

$PR_i^1 = d* \sum i_j^0 * w_{ji} + (1-d)* P_i$

Example: second iteration



= i2_1*d*w21=(1-d)*d*w21

= i2_1*d*w23= (1-d)*d*w23

= i3_1* d * w34=(1*d*w23)*d*w34

= i1_1 *d * w13=(1*d*w21)*d*w13

$d* \sum i_j^1 * w_{ji}$

$PR_i^2 = d* \sum i_j^1 * w_{ji} + (1-d)* P_i$

S	D	w
1	3	w13
2	1	w21
2	3	w23
3	4	w34
4	1	w41
4	2	w42

graph

id	i
1	i1_1
3	i3_1
2	i2_1

i2_1

i1_1

i3_1

Second iteration

D	next-i
1	i2_1* d *w21
3	i2_1*d*w23+i1_1*d* w13
4	i3_1* d *w34

id	i
1	i2_1* d *w21
3	i2_1*d*w23+i1_1*d*w13
4	i3_1* d *w34
2	(1-d)*1

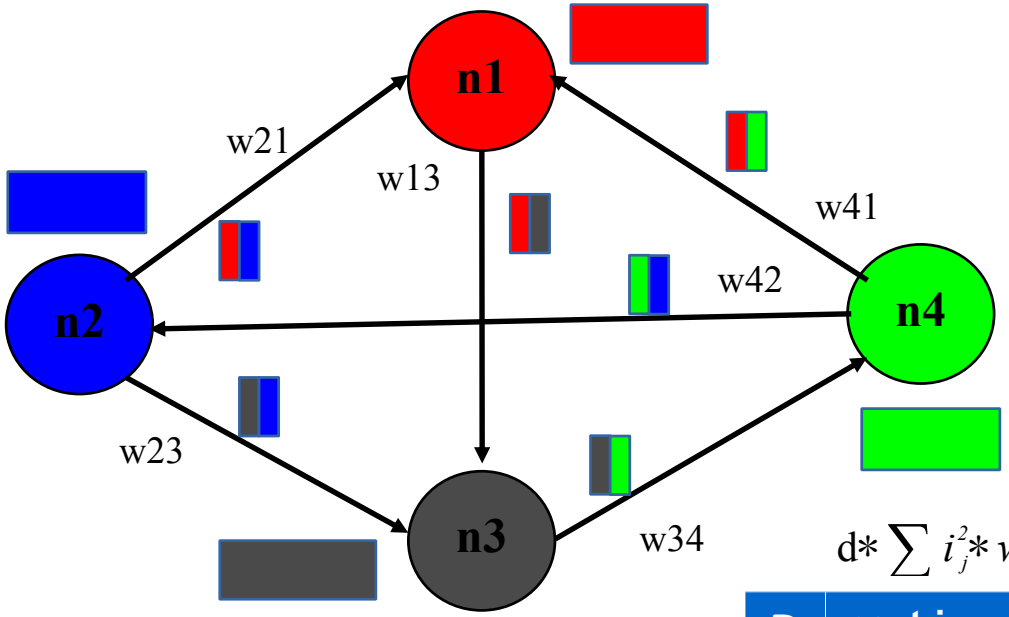
i2_2= 1-d

i1_2= i2_1* d *w21

i3_2= i2_1*d*w23 +i1_1*d*w13

i4_2= i3_1* d *w34

Example: third iteration



= i2_2*d*w21

= i2_2*d*w23

= i3_2* d * w34

= i1_2*d * w13

= i4_2*d * w41

= i4_2*d * w42

$d* \sum i_j^2 * w_{ji}$

$PR_i^3 = d* \sum i_j^2 * w_{ji} + (1-d)* P_i$

S	D	w
1	3	w13
2	1	w21
2	3	w23
3	4	w34
4	1	w41
4	2	w42

graph

id	i
1	i1_2
3	i3_2
2	i2_2
4	i4_2

Third iteration

D	next-i
1	i2_2* d **w**21+i4_2d*w41
3	i2_2* d*w23+i1_2*d*w13
4	i3_2* d *w34
2	i4_2* d *w42

id	i
1	i2_2* d **w**21+i4_2d*w4
3	i2_2* d*w23+i1_2*d*w13
4	i3_2* d *w34
2	i4_2* d *w42+(1-d)*1

i2_3= i4_2*d*w42+(1-d)*1

i1_3

i3_3

i4_3

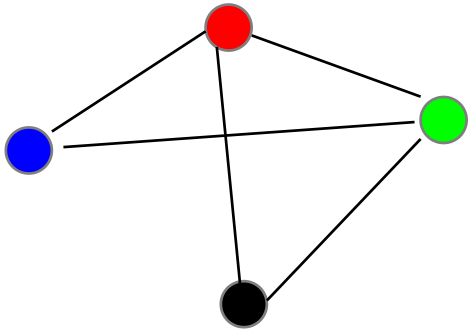
Algorithms for community detection

Clustering coefficient

Clustering coefficient *for* an undirected graph G=(V,E):

- measure of grouping nodes in a graph, used in small world networks (social networks), community detection
- **cc(v)** = fraction of neighbors of v which are themselves neighbors

$$= \frac{|\{(u,w) \in E | u \in \Gamma(v) \wedge w \in \Gamma(v)\}|}{\binom{d_v}{2}}$$



CC(●) = 1/1

CC(●) = 2/3

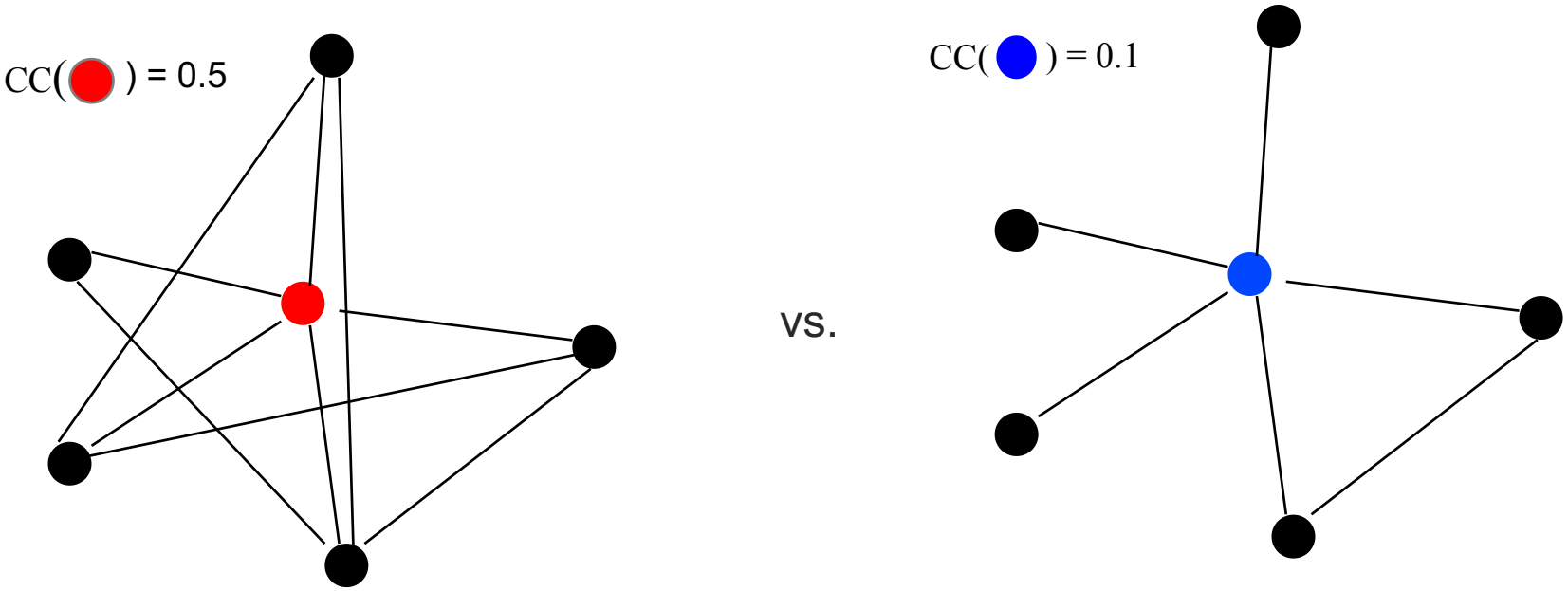
CC(●) = 1/1

CC(●) = 2/3

Clustering coefficient : the probability that a node's neighbors are connected to each other

Clustering coefficient

Shows the density of connectivity around a node

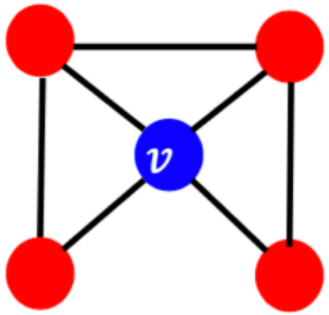


Applications:

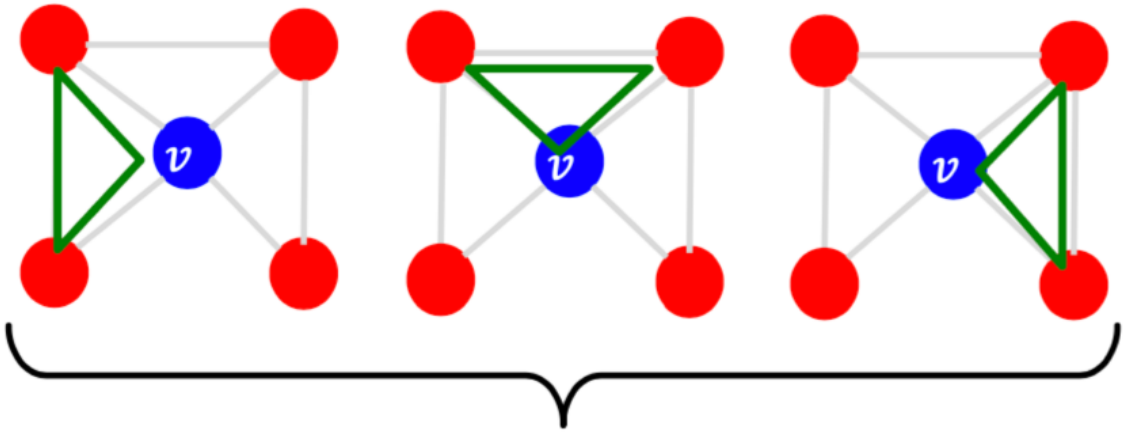
- study the community structure of the Facebook social graph (find dense neighborhoods of users in the global sparse graph)
- explore the thematic structure of the Web: detect page communities/common topics on the basis of reciprocal links.

Counting triangles

- Clustering coefficient: Represents the number of triangles in the graph



$e_v = 0.5$



3 triangles out of the 6 possible triples

How to count triangles?

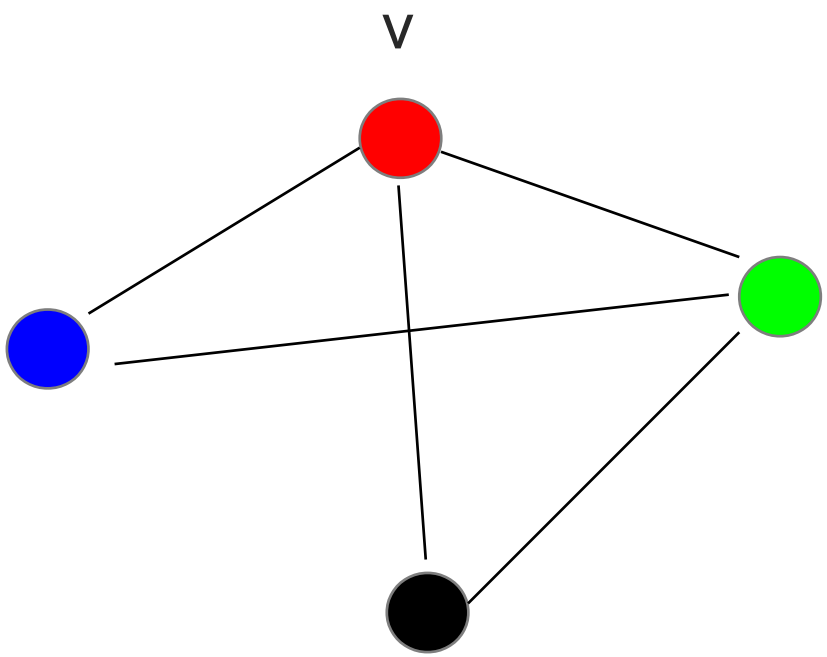
Sequential algorithm (undirected graph):

```
NbTriangles ← 0
foreach node v:
  foreach pair u,w in Γ(v)
    if (u,w) is an edge
      NbTriangles += 1/2
return (NbTriangles / 3)
```

Complexity of the algorithm: $O\left(\sum d_v^2\right)$

Each triangle is counted 3 times (once per node)

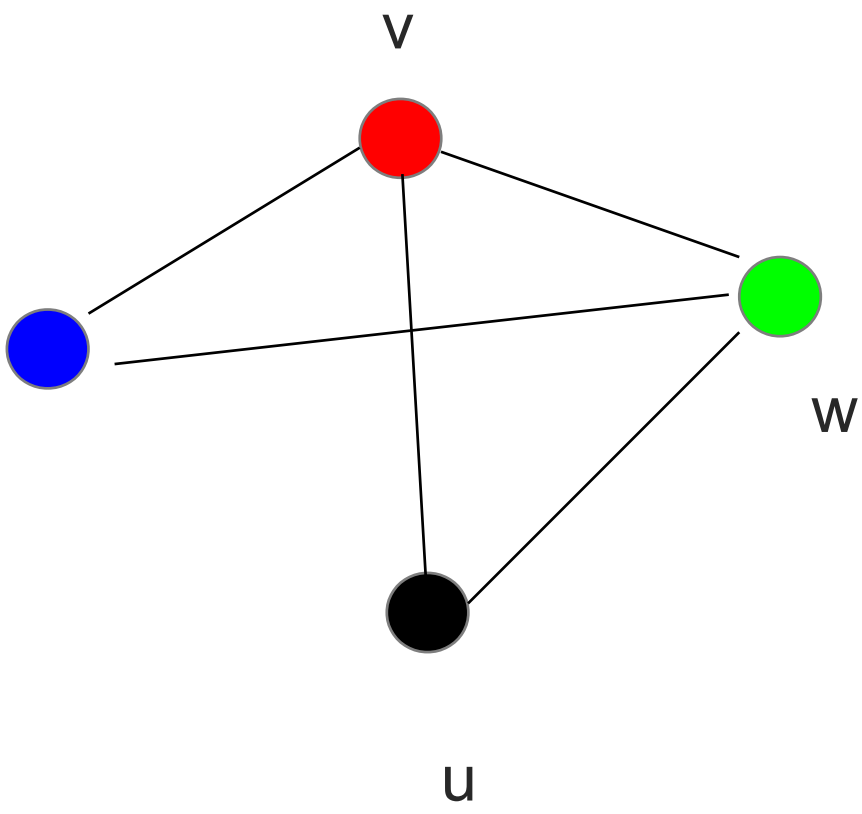
Example: computation of triangles



From node v

NbTriangles = 0

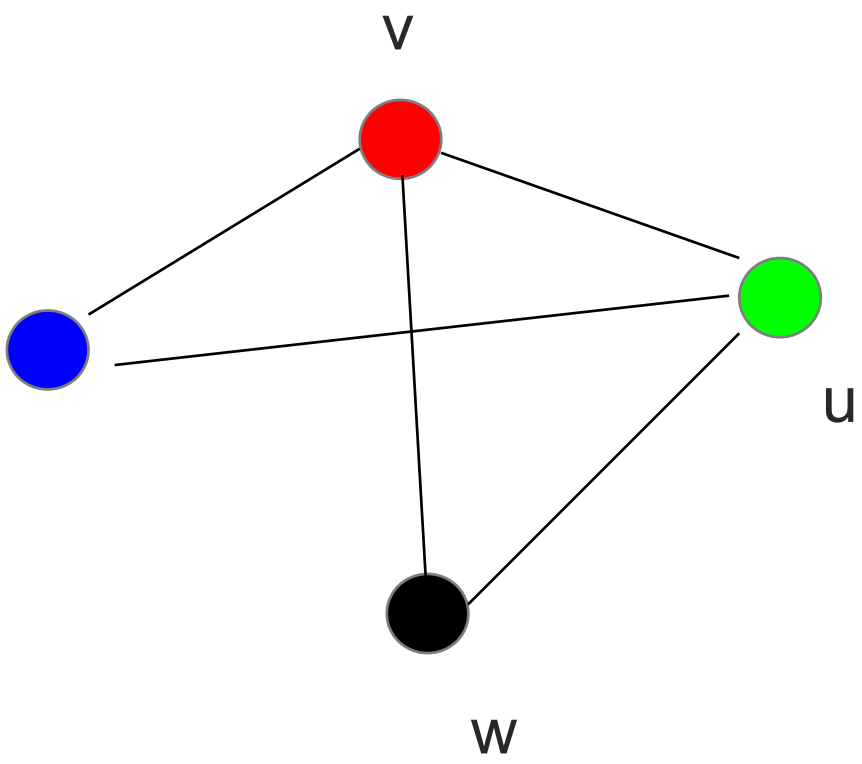
Example



From node v

NbTriangles = 1/2

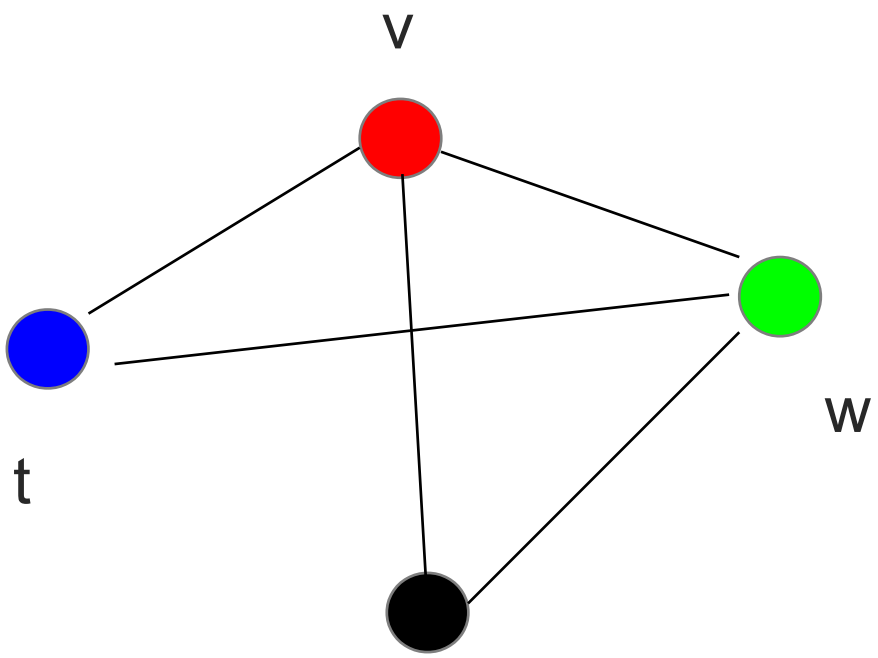
Example



From node v

NbTriangles = 1

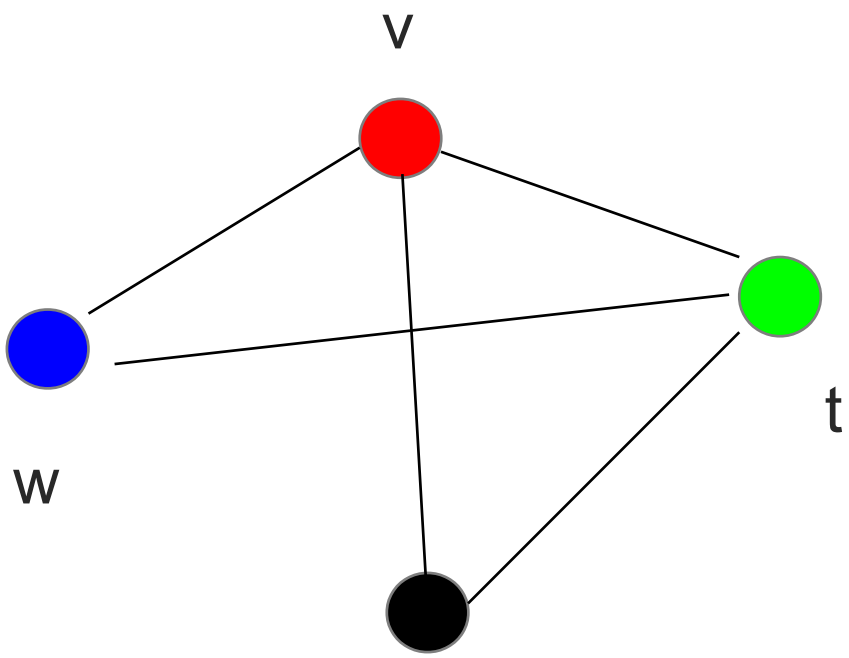
Example



From node v

NbTriangles = 1.5

Example



From node v

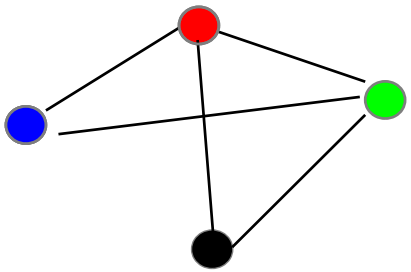
NbTriangles = 2

DataFrame algorithm

Step 1 : Input: $\{(v,u)|\ \}_{u \in \Gamma(v)}$

foreach $u \in \Gamma(v)$ emit {(v,u)}

Example : (●●)(●●)(●●)(●●)(●●)(●●)(●●)



Step 2 : Input: $\{(v,u) \mid u \in \Gamma(v)\ \}$

foreach(u,w): $u,w \in \Gamma(v)$

emit {((u,w), v)} //all possible edges in the graph

Example : ((●●)●) ((●●)●) ((●●)●),.....

Step 3 : emit { ((u,w), $)| $w \in \Gamma(u)$ }

Step 4 : Input : {(u,w)| v1,v2,...vk,$?}

foreach (u,w) if $ is part of the input, then :

NbTriangles[vi] +=1/2

((●●)● $) → NbTriangles(●) +1/2

(●●)● → ∅

Algorithm adaptation

- We generate all the paths to check in parallel, the execution time is
- $max_{v \in V}(\sum d_v^2)$ => for nodes with many neighbors (millions) the corresponding reducer tasks can be very slow

Improvement :

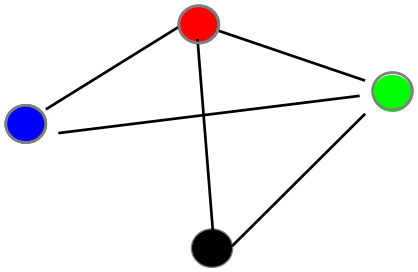
- order the nodes by their degree (for those which have the same degree by their identifier)
- count each triangle only once, starting from the minimum node
- complexity: $O(m^{3/2})$ (m = number of arcs in the graph)

Improved algorithm

Sequential algorithm:

```
nbTriangles ← 0
foreach v in V
    foreach u,w in Adjacency(v)
        if u > v && w > u
            if (u,w) in E
                nbTriangles++
return nbTriangles
```

DataFrame algorithm



Step 1 : Input: $\{(v,u)|\ u \in \Gamma(v)\}$

if u > v then emit {(v,u)}

Example : (● ●) (● ●) (● ●) (● ●) (● ●)

Step 2 : Input : $\{(v,u)\ |\ u \in S \subset \Gamma(v)\}$

foreach (u,w) : $u,w \in S$ //u,w: neighbors of v (v< u et v< w) ● < ● < ● < ●

if w > u then emit {((u,w), v)}

Example : ((● ●)●) ((● ●)●) ((● ●)●))

Step 3 : emit $\{((u,w), \$)|_{w \in \Gamma(u)}\}$, u < w

Step 4 : Input : {(u,w)| v1,v2,...vk,$?}

foreach (u,w) if $ is part of the input, then : NbTriangles[vi] ++

Example :((● ●)● $) → NbTriangles(●)++

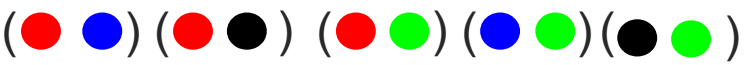
(● ●)● → ∅

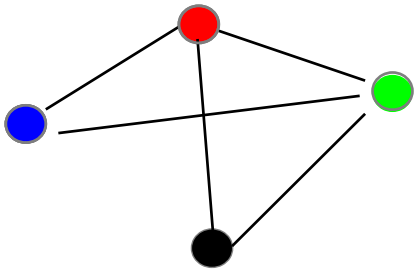
Computation with DataFrames

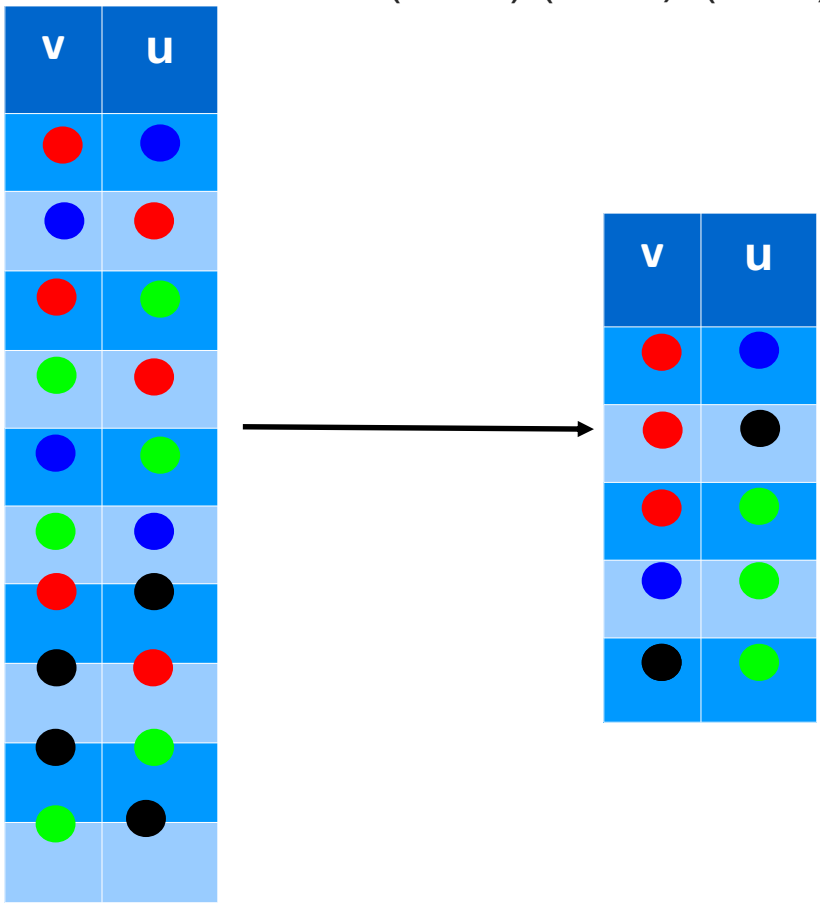
Step 1 : Input : $\{(v,u) | u \in \Gamma(v)\}$

if u > v then emit {(v,u)}

Example :







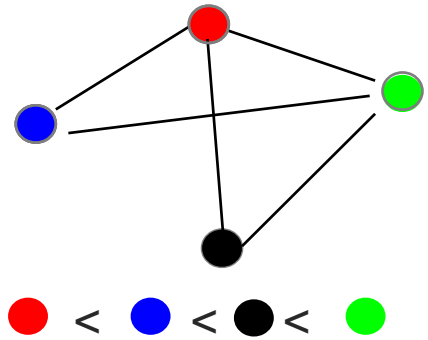
Computation with DataFrames

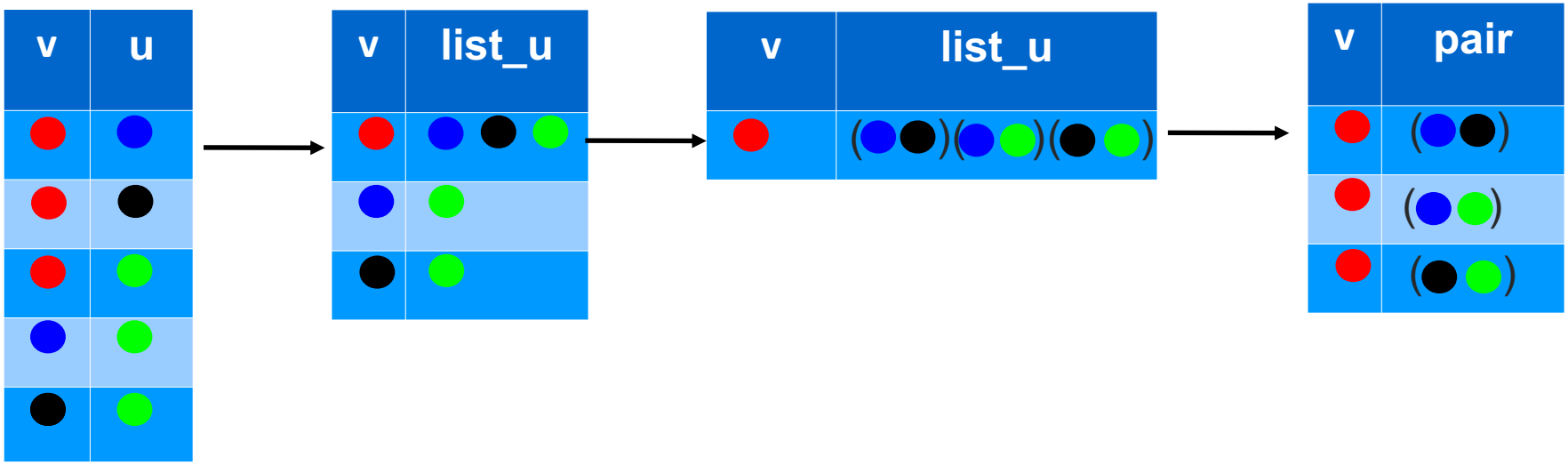
Step 2 : Input : $\{(v,u) \mid u \in S \subset \Gamma(v)\}$

foreach (u,w) : $u,w \in S$ //u,w: neighbors of v (v< u et v< w)

if w > u then emit {((u,w), v)}

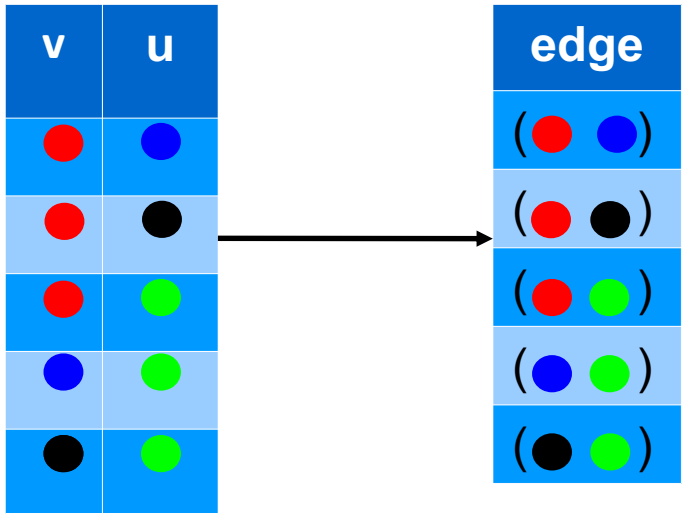
Examples: ((● ●)●)((● ●)●) ((● ●)●))

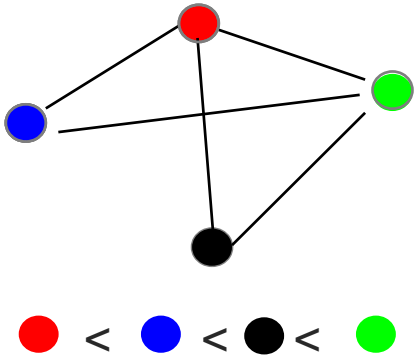




Computation with DataFrames

Step 3 : emit { ((u,w), $)| $w \in \Gamma(u)$ }, u < w

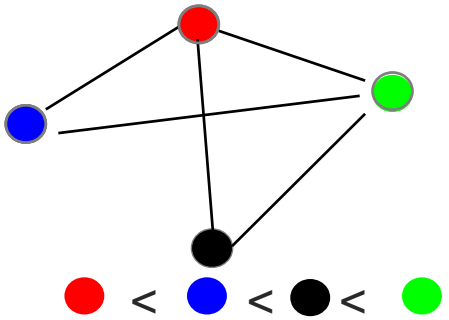


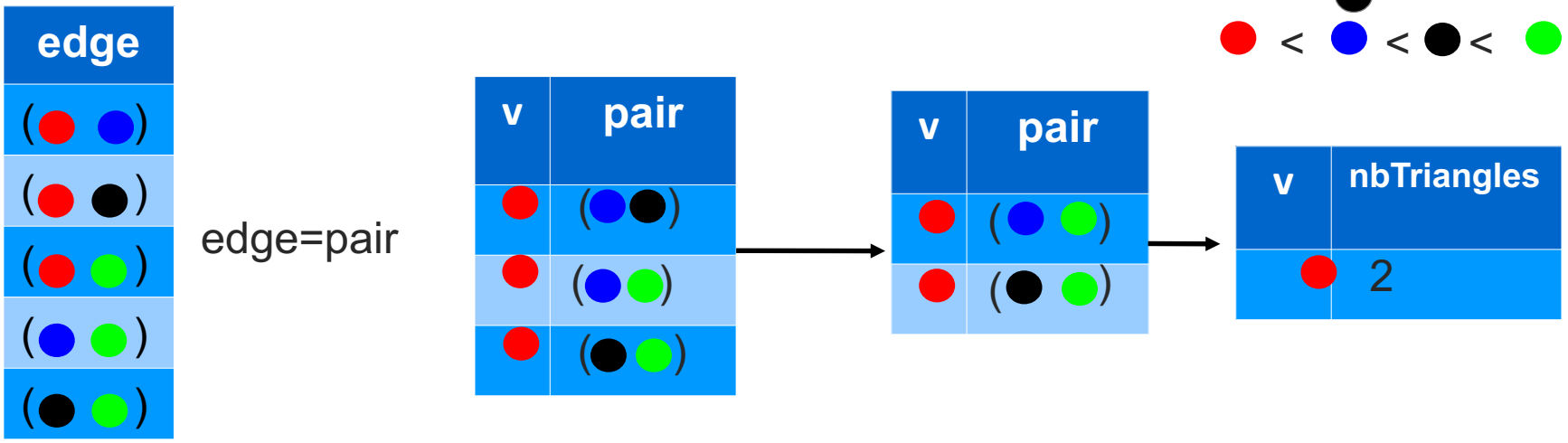


Computation with DataFrames

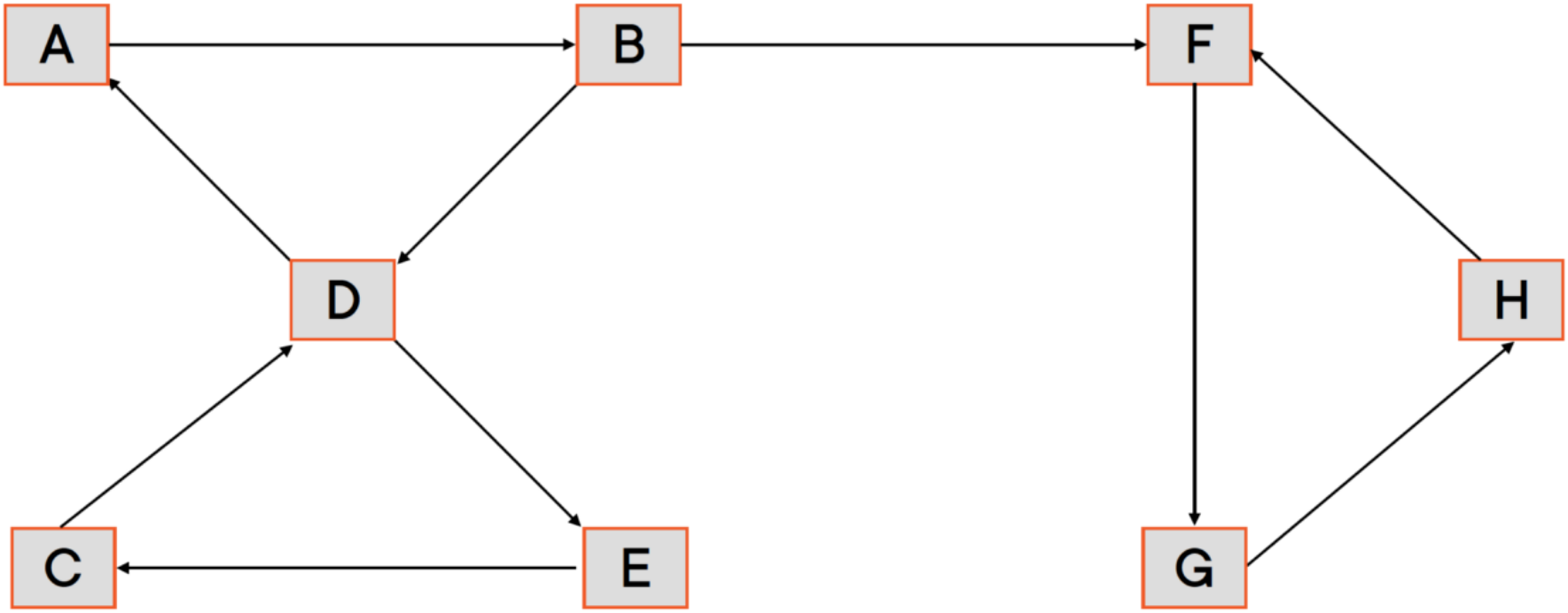
Step 4 : Input : {(u,w)| v1,v2,...vk,$?}

foreach (u,w) if $ is part of the input, then : NbTriangles[vi] ++





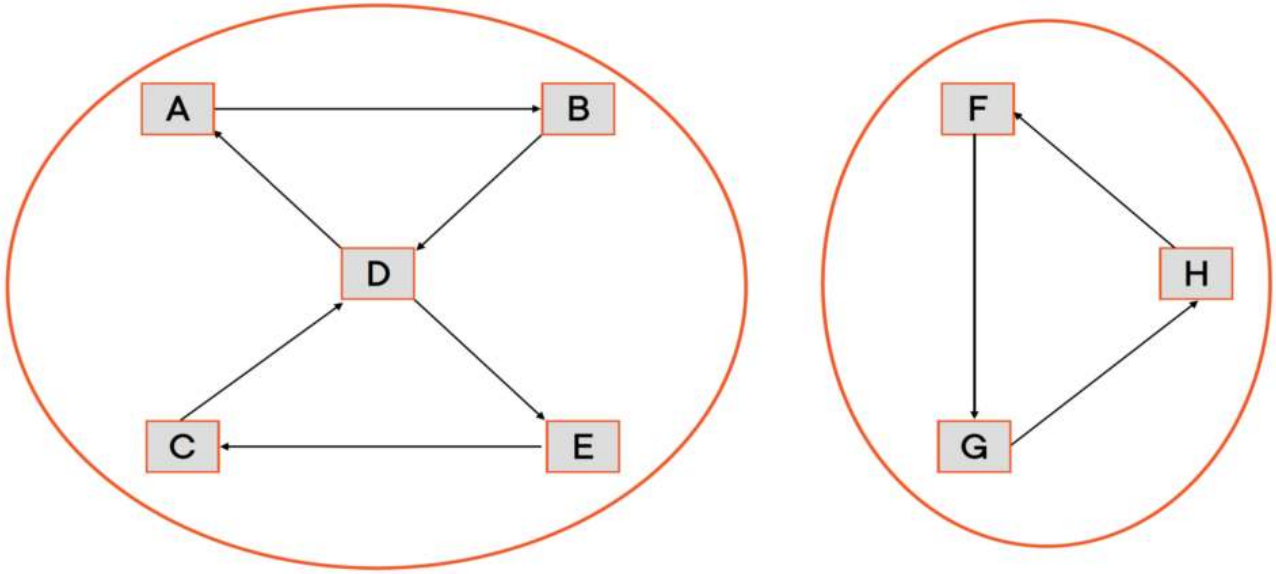
Strongly connected components



- There is a path between each pair of vertices (maximum strongly connected subgraph)

Strongly connected components

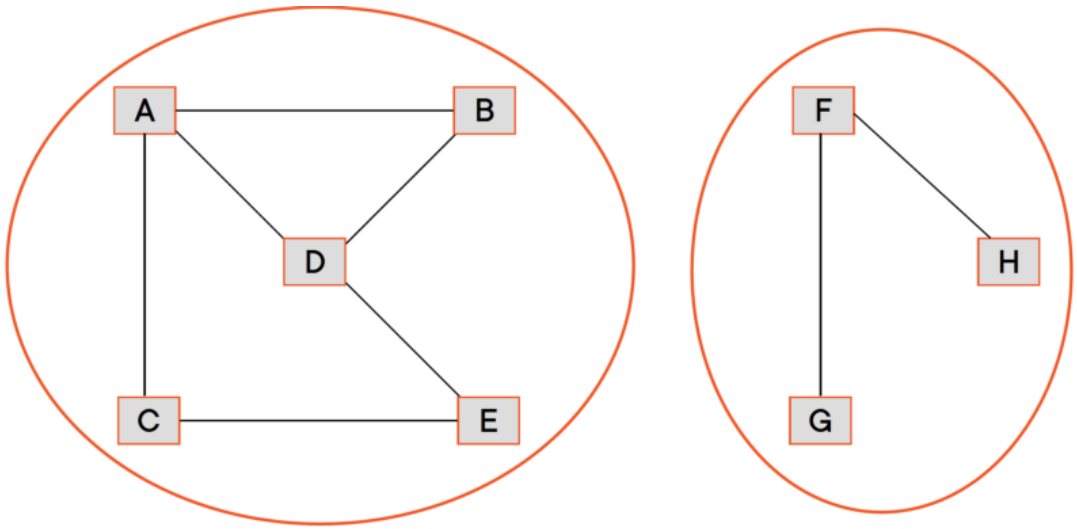
- Two strongly connected components



- Applications:
  - find sets of companies in which each member directly and/or indirectly owns shares of all other members
  - calculate the connectivity of different network configurations when measuring routing performance in multi-hop wireless networks.

Connected Components: Undirected Graph

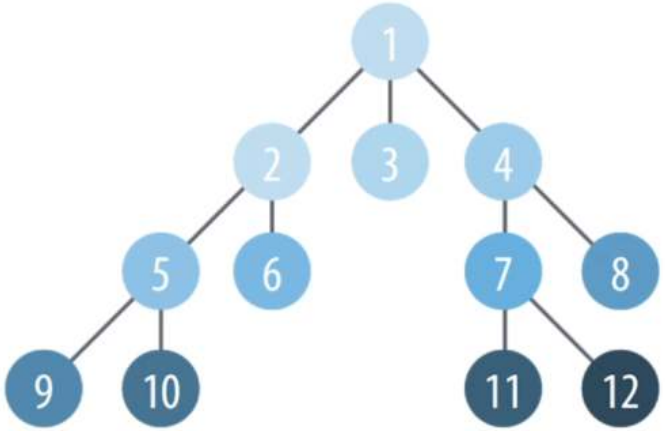
- Set of vertices connected in pairs by a path (maximum connected subgraph) which are not connected to other nodes in the graph
- If there is a path between nodes, they must belong to the same component



- Applications:
  - Analysis of citation networks
  - determine the degree of connectivity of a network (see if connectivity maintained if one removes a central node or an authority node)

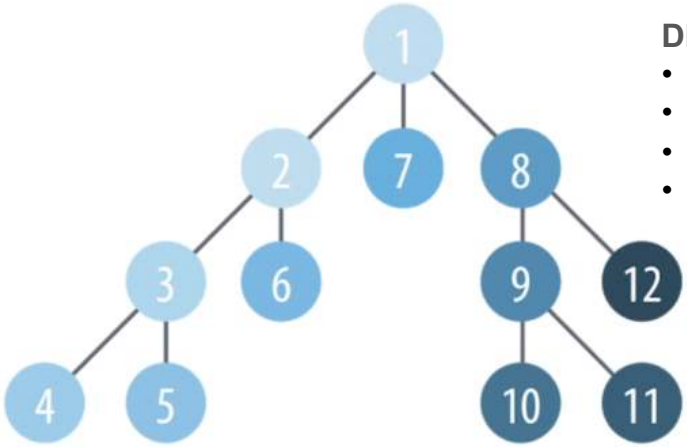
Algorithms for path finding and graph exploration

BFS and DFS



BFS (Neo4J and Spark)

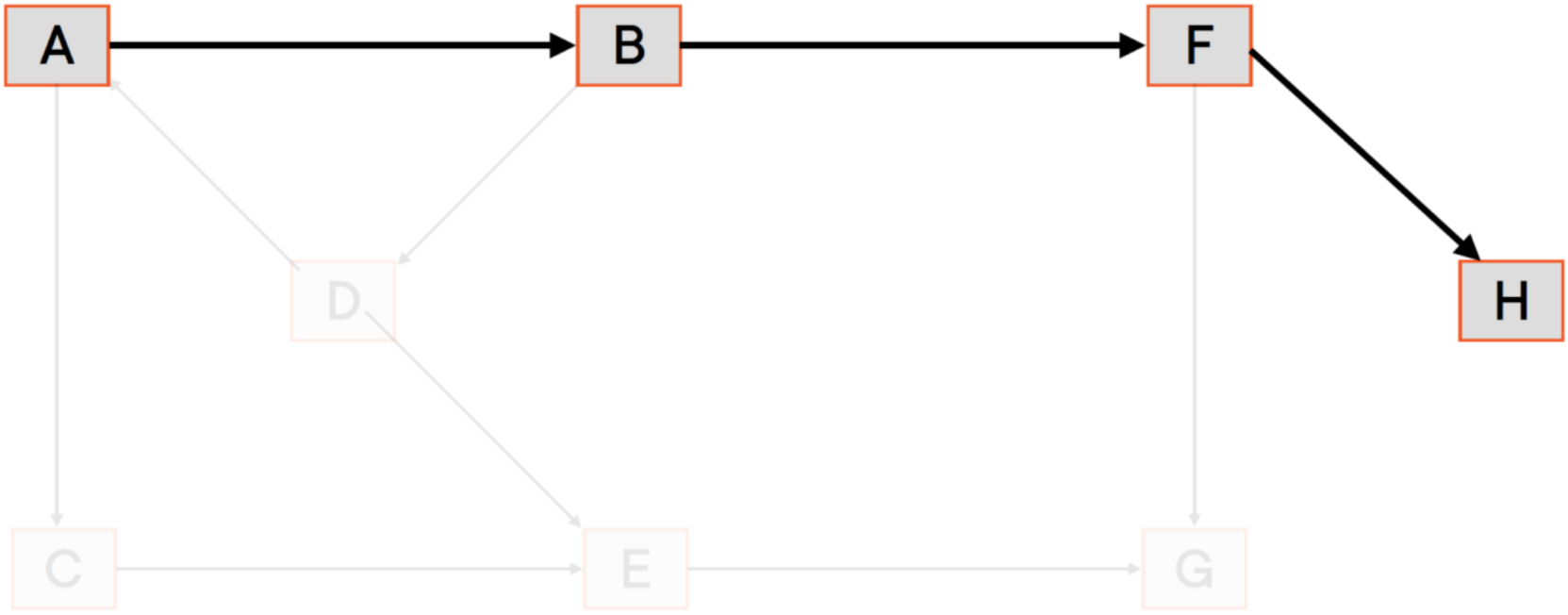
- Choose a source node (1)
- Explore all neighbors as far as possible in a chosen order (2, 3, 4)
- Visit neighbors' neighbors (5,6,7, 8....)
- Stop: the entire graph has been visited or a specific criterion is satisfied



DFS

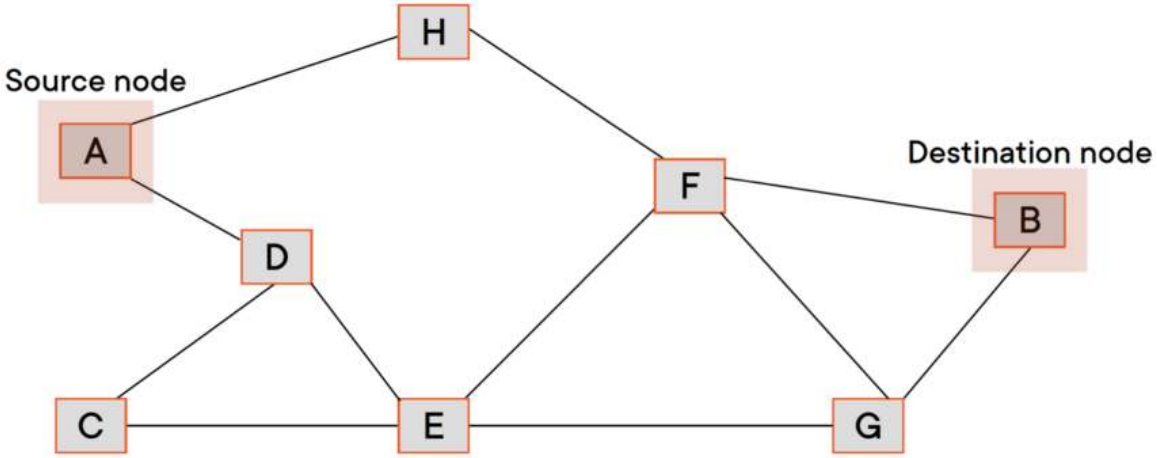
- Choose a source node (1)
- Choose a neighbor, visit one of your neighbors, etc. , as far away as possible
- Choose another neighbor and start again: 1, 2, 3, 4, 5, 6, 7...
- Stop: the entire graph has been visited or a specific criterion is satisfied

Path in a directed graph



Sequence of consecutive arcs connecting node A to node H (the path follows the direction of the arcs)

Shortest paths

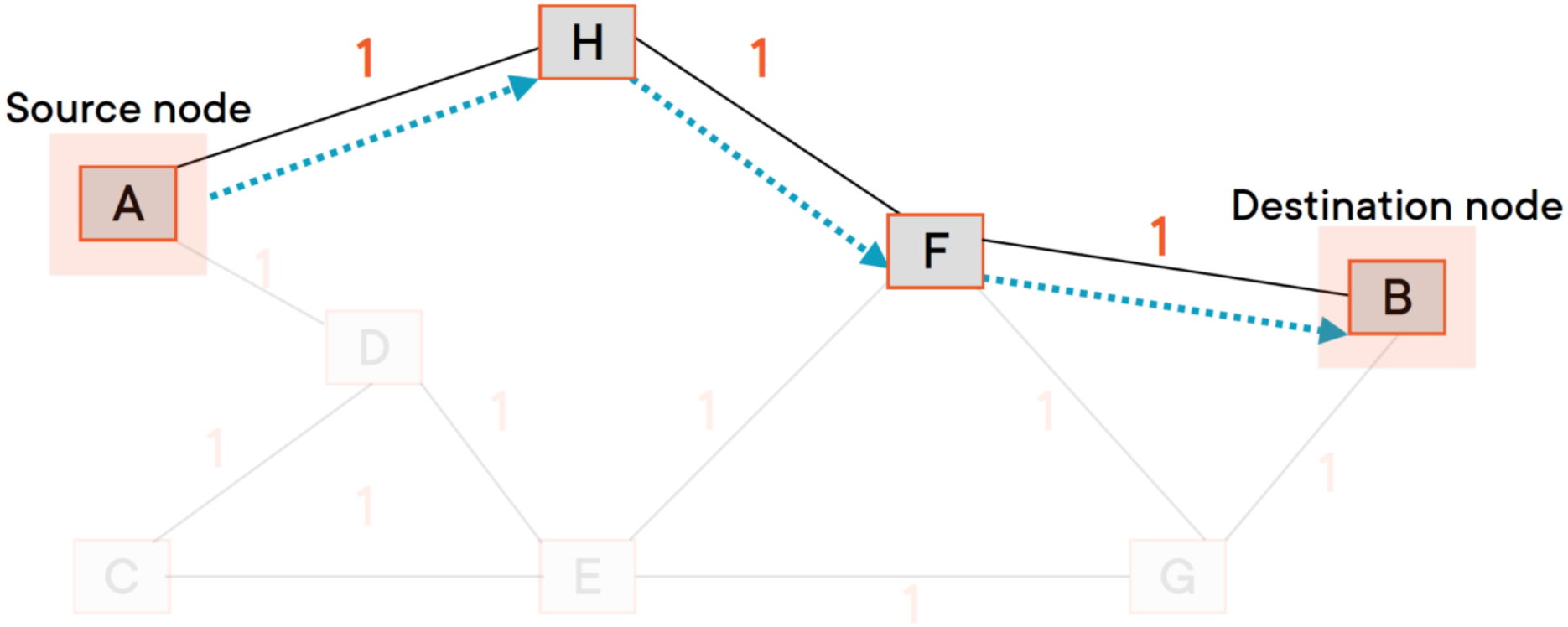


- Find the shortest path between a source node and a destination node
  → depends on the weight of the edges

Applications:

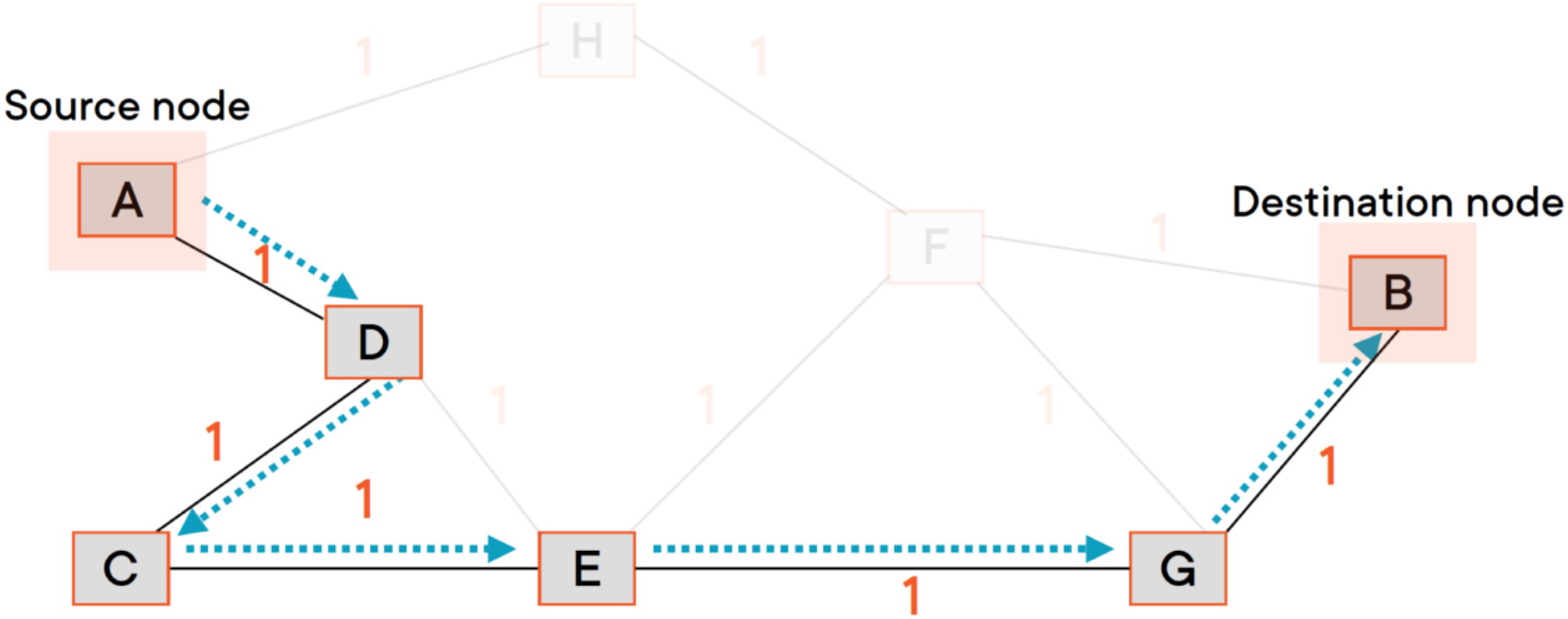
- Kevin Bacon number (number of degrees of separation between actors and Kevin Bacon, based on the films they starred in)
- LinkedIn uses the shortest path algorithm
- Google Maps : Find Routes Between Places

Unweighted graph



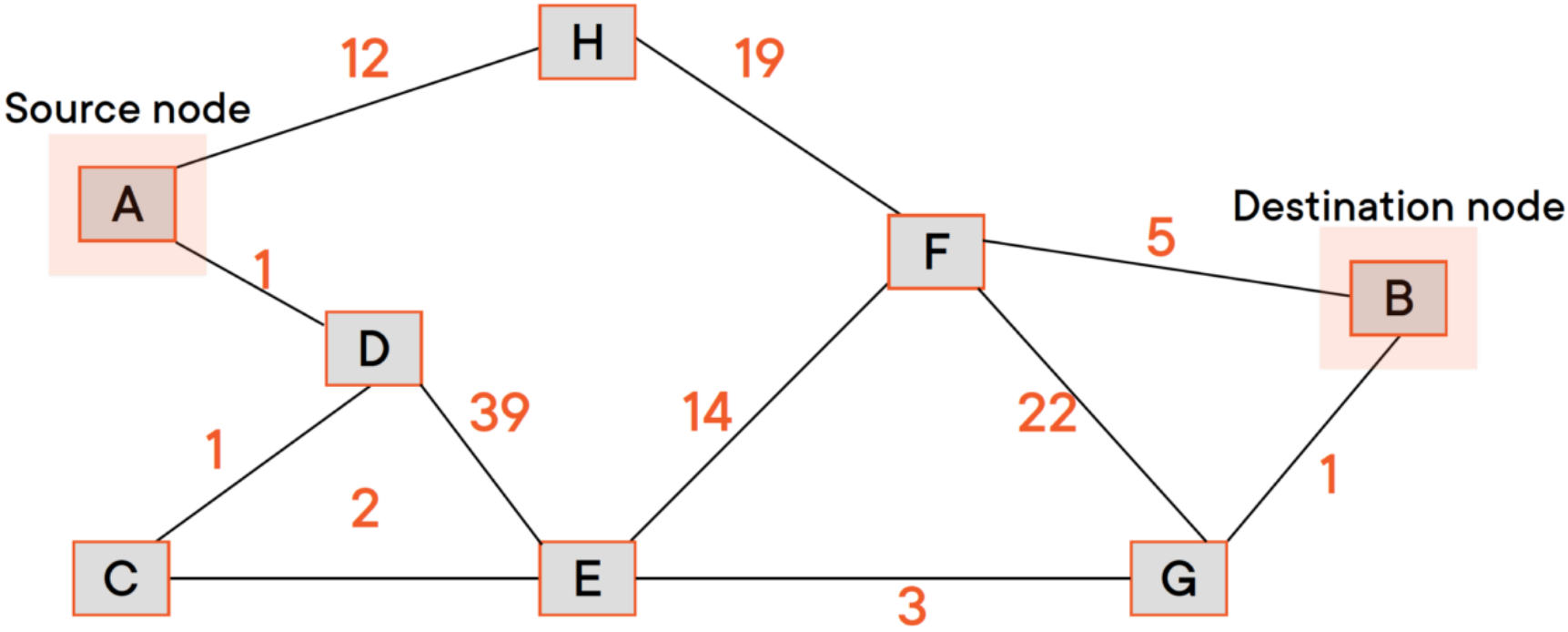
- All edges have the same weight (=1) → shortest path with minimum number of vertices

Unweighted graph



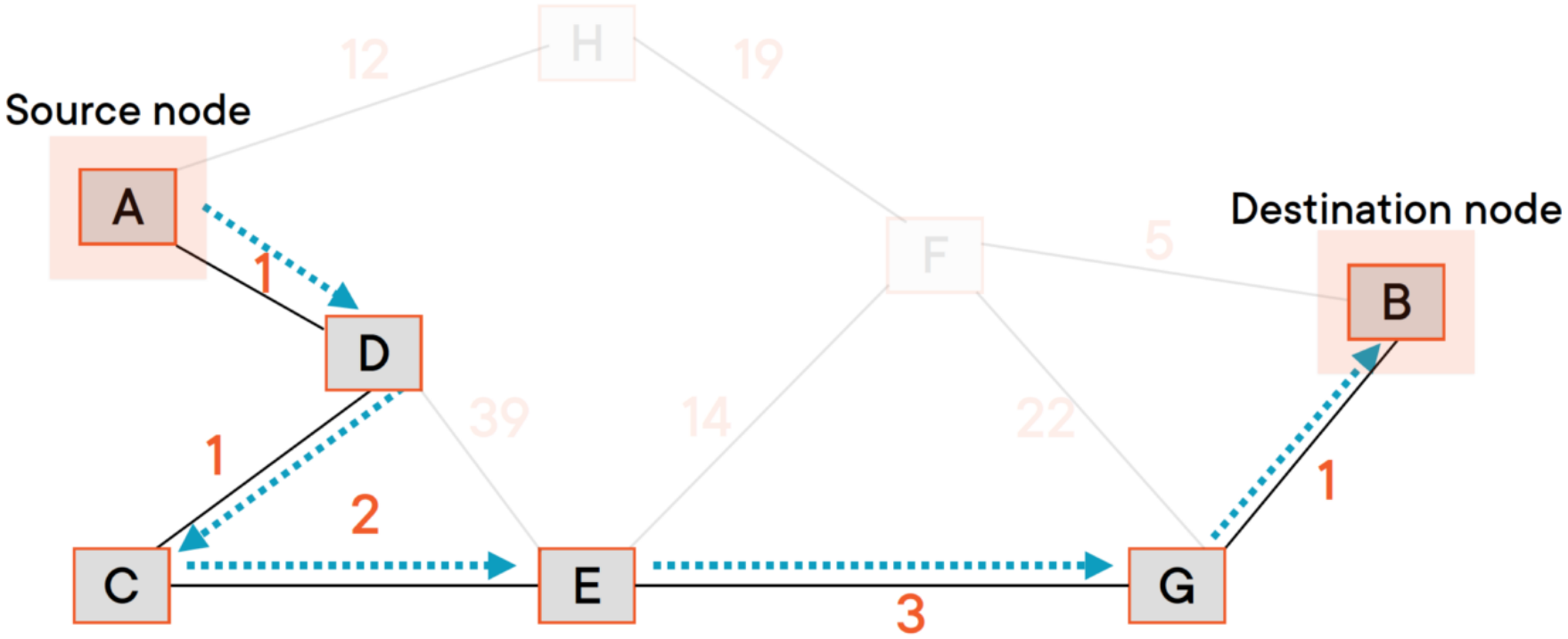
- There are other, longer paths (5)

Weighted graph



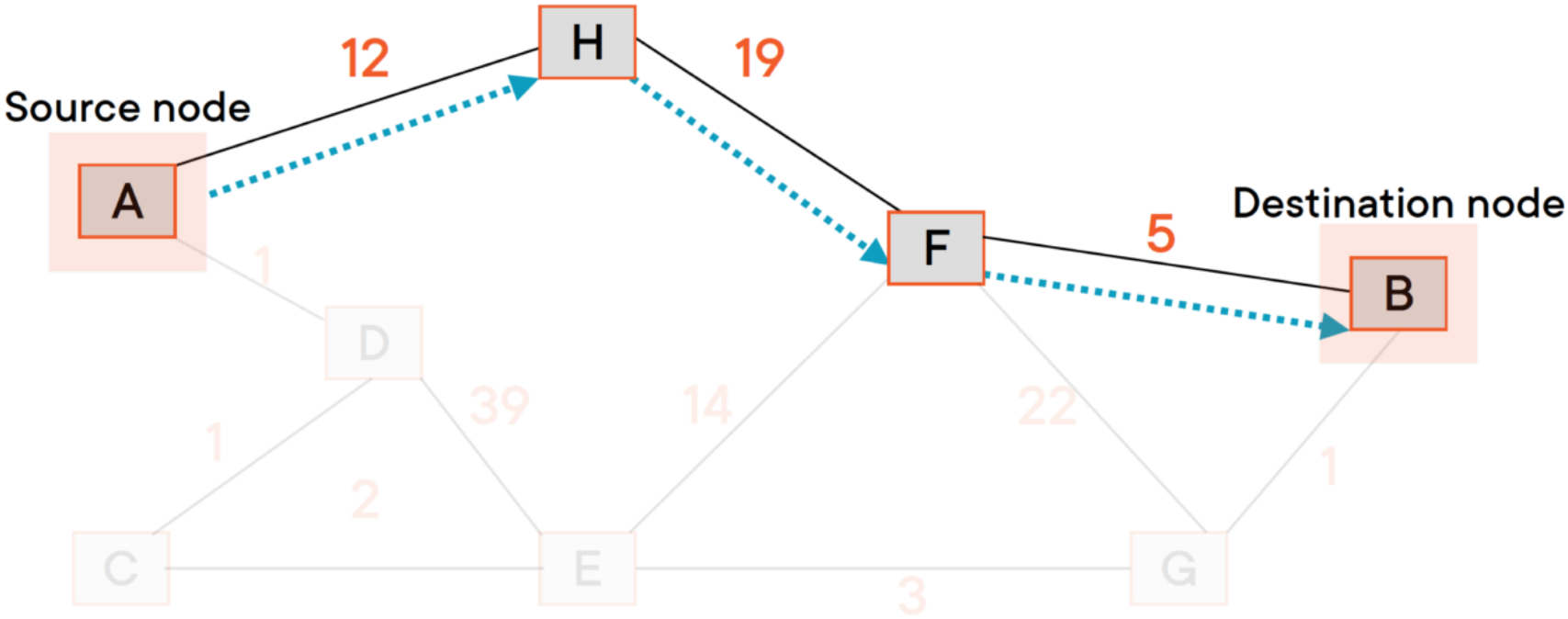
- Shortest path → depends on the sum of edge weights (Dijkstra algorithm)

Weighted graph



- Shortest path cost: 1+1+2+3+1 = 8

Weighted graph



- Path cost: 12+19+5 = 36

Shortest paths

Computation

Find the length of the shortest path from a given node **s** to other nodes

Centralized: Dijkstra's algorithm

MapReduce: **Parallel BFS** from initial node s

Intuition (for an unweighted graph):

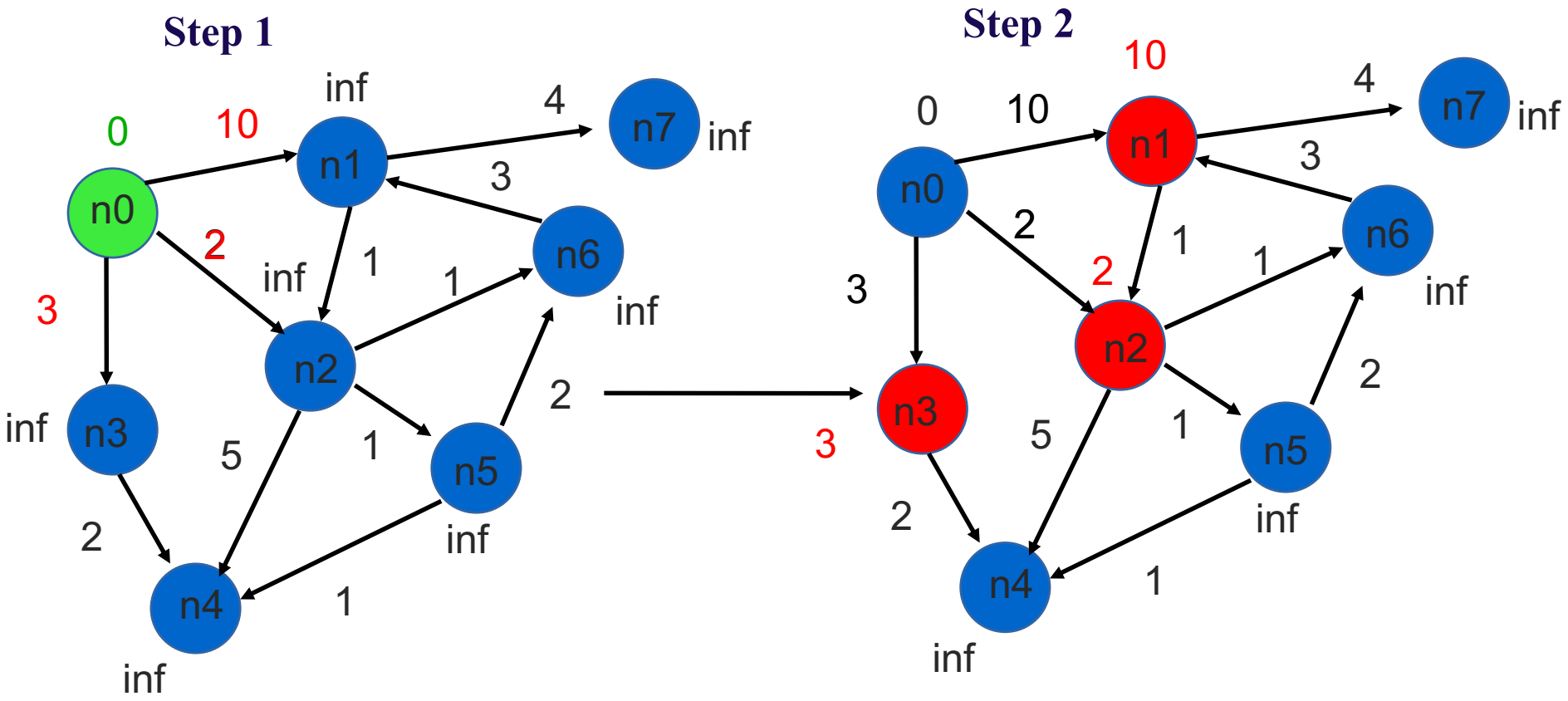
- For all neighbors p of s:

  DISTANCE(p) = 1

- For all nodes n reachable from a set of nodes M

  $$\text{DISTANCE}(n) = 1 + \min_{m \in M} (\text{DISTANCE}(m))$$

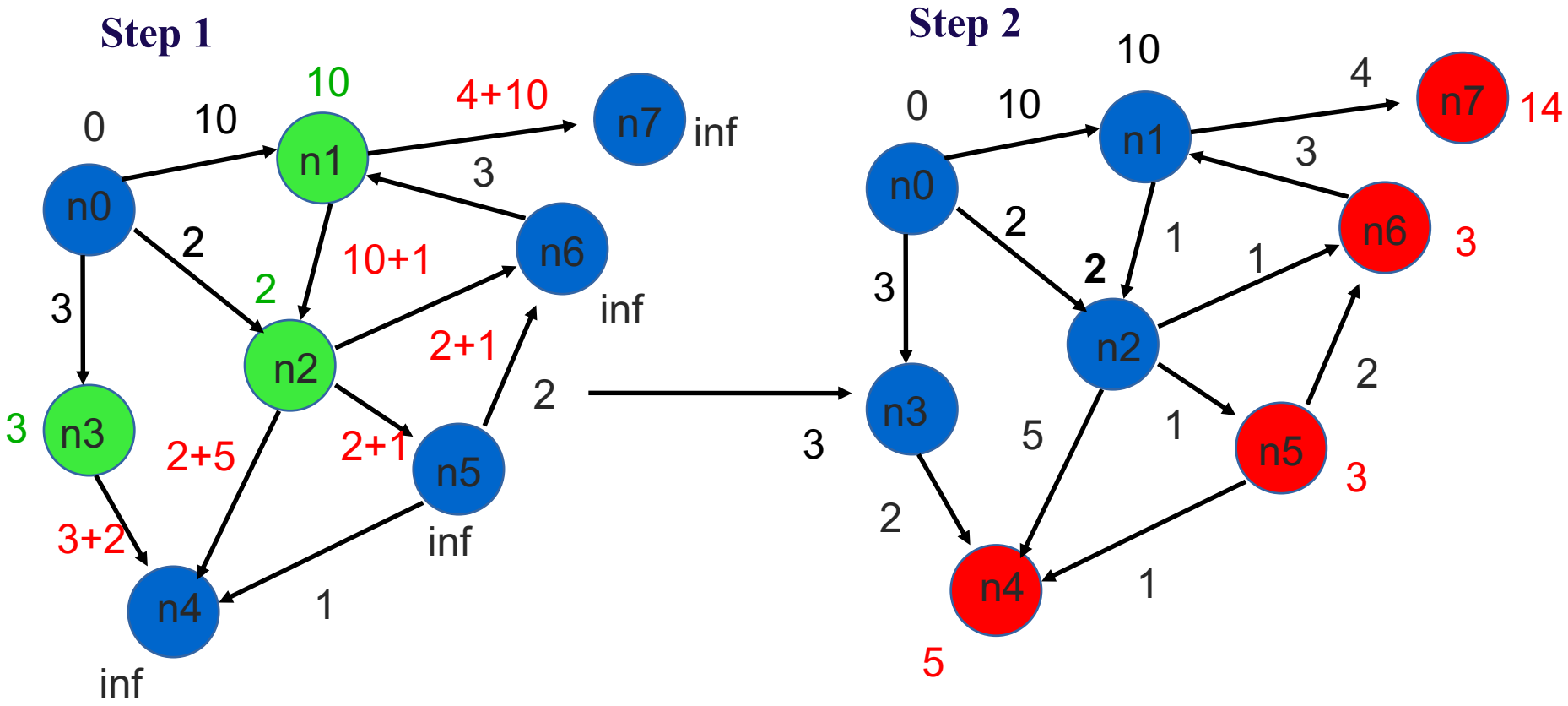
BFS: weighted graph



First iteration:

- Active node: n0

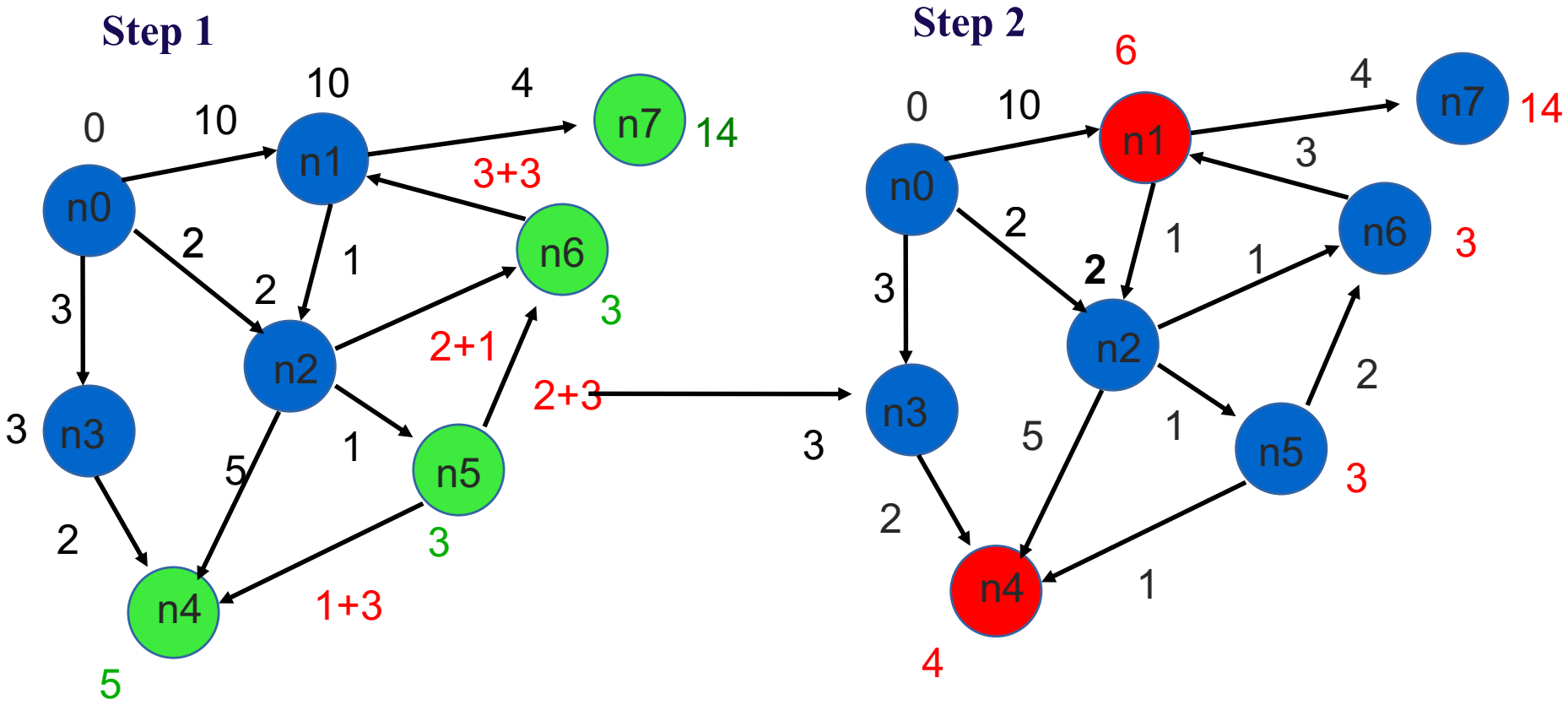
BFS: weighted graph



Second iteration:

- Active nodes: n1, n2, n3

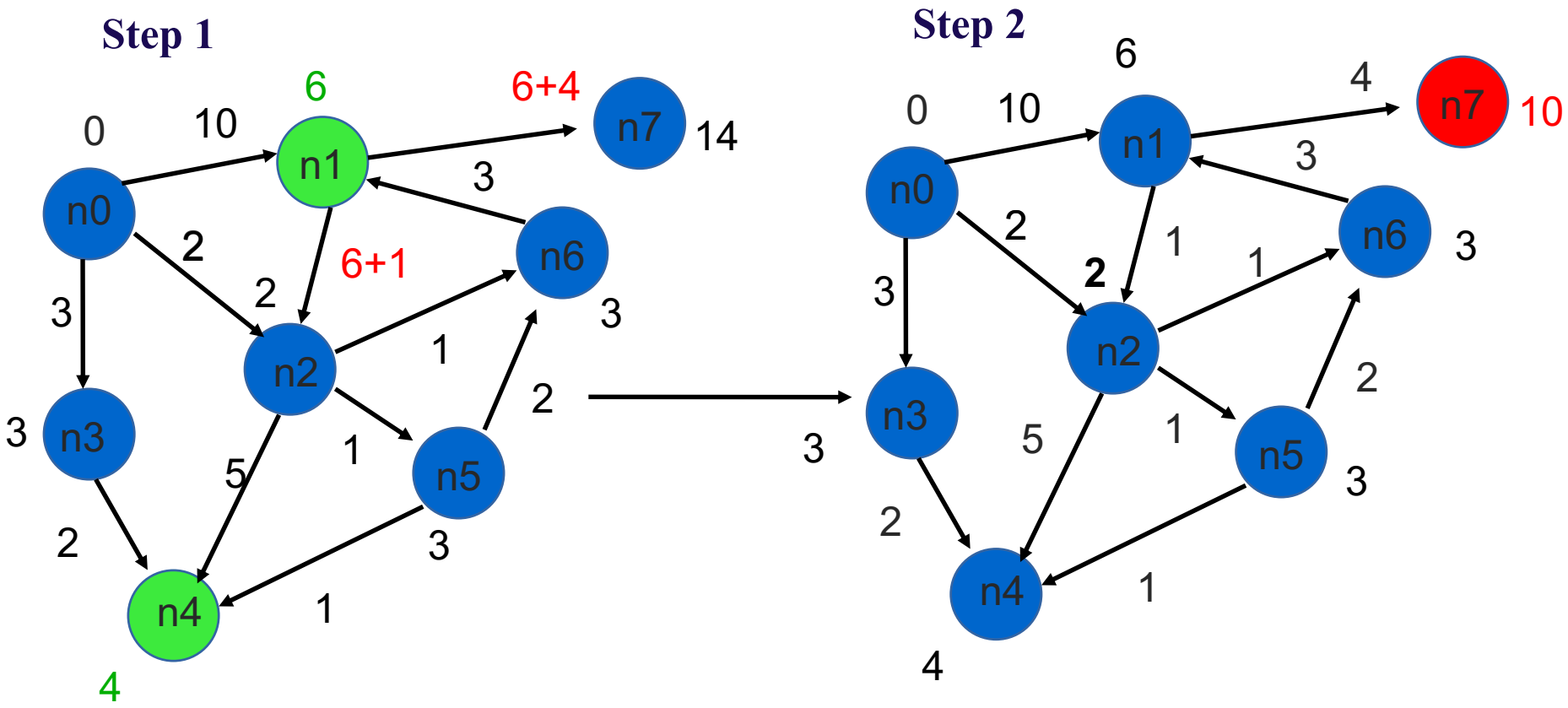
BFS: weighted graph



Third iteration:

- Active nodes: n4, n5, n6, n7

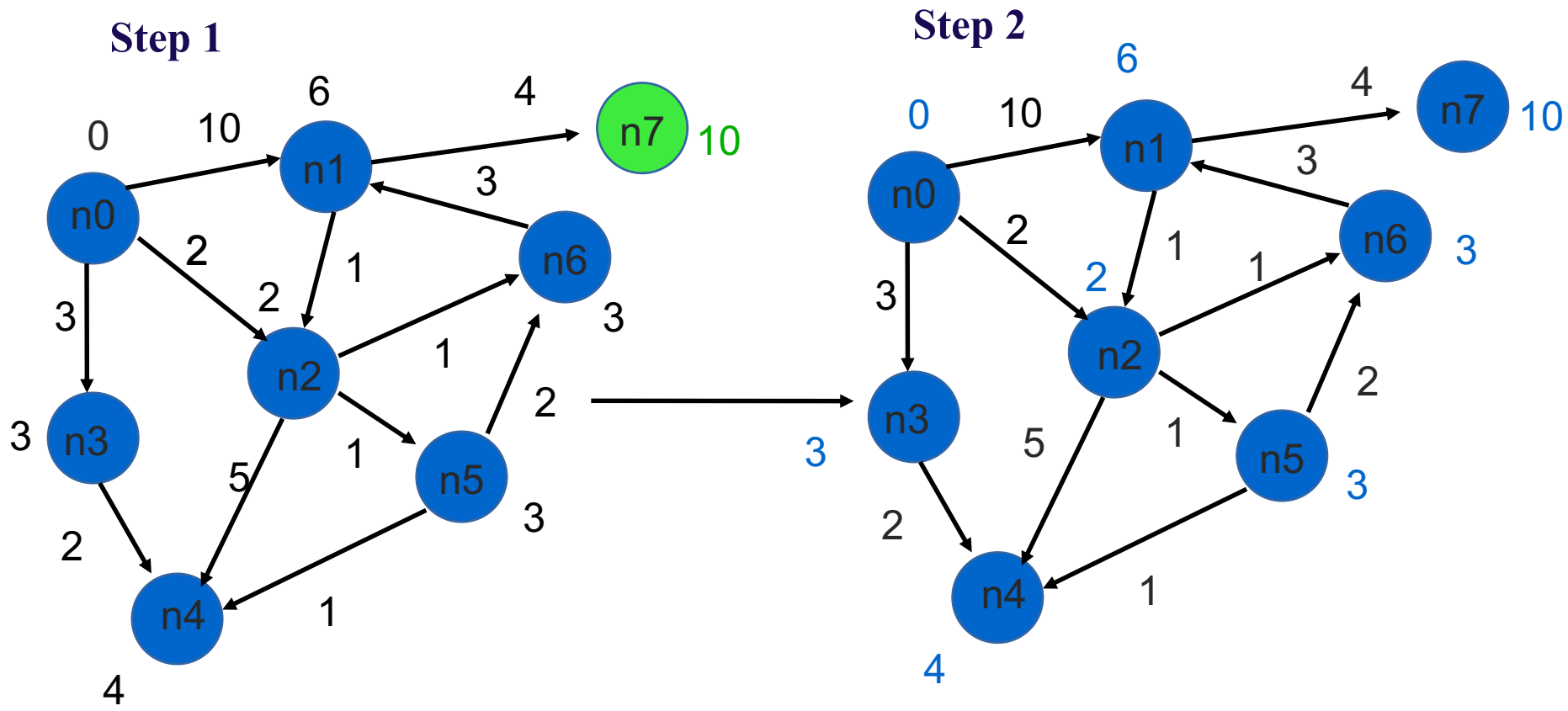
BFS: weighted graph



Fourth iteration:

- Active nodes: n4, n1

BFS: weighted graph



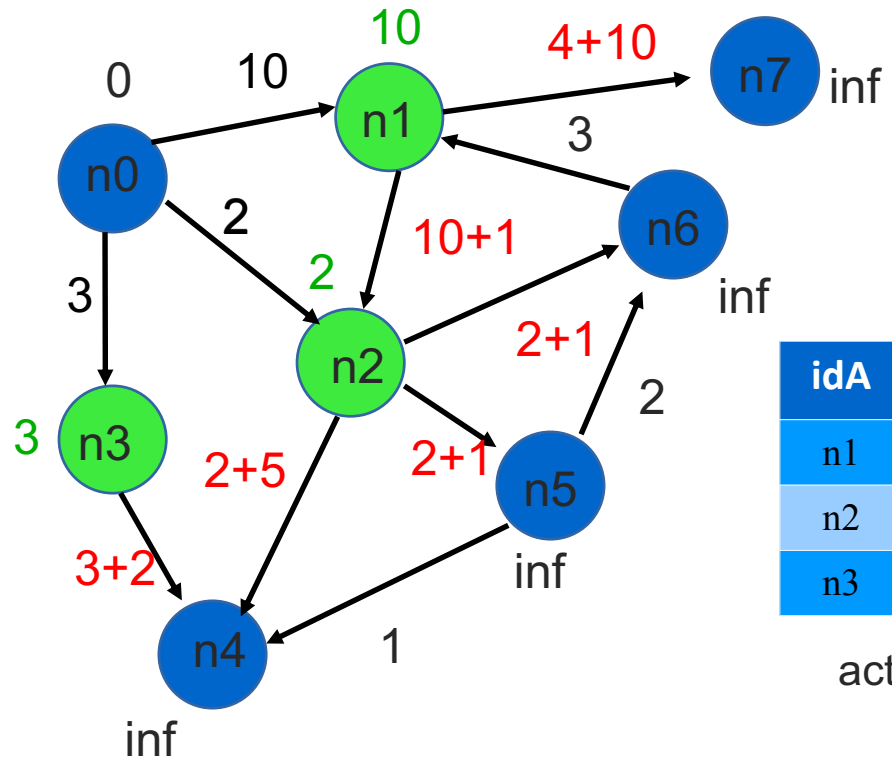
Fifth iteration:

- Active node: n7

In summary:

- A node becomes active when its distance has been updated
- Stop the computation when there are no more active nodes

BFS : Exemple avec DataFrame (Step 1)



Second iteration

idA	d
n1	10
n2	2
n3	3

active

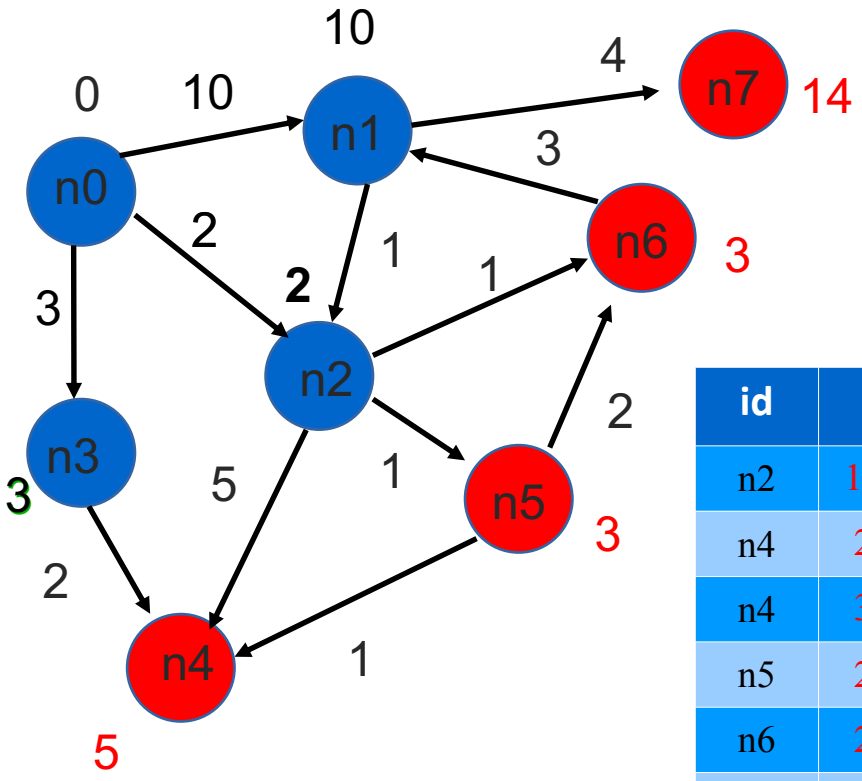
S	D	w
n0	n1	10
n0	n3	3
n0	n2	2
n1	n7	4
n1	n2	1
n2	n4	5
n2	n5	1
n2	n6	1
n3	n4	2
n5	n4	1
n5	n6	2
n6	n1	3

graph

id	d
n2	10+1
n4	2+5
n4	3+2
n5	2+1
n6	2+1
n7	4+10

new_distances

BFS: Example with DataFrame (**Step 2**)



Second iteration

id	d
n2	10+1
n4	2+5
n4	3+2
n5	2+1
n6	2+1
n7	4+10

new_distances

id	d
n0	0
n1	10
n2	2
n3	3
n4	inf
n5	inf
n6	inf
n7	inf

prev_distances

id	d
n0	0
n1	10
n2	**2**
n3	3
n4	5
n5	3
n6	3
n7	14

distances