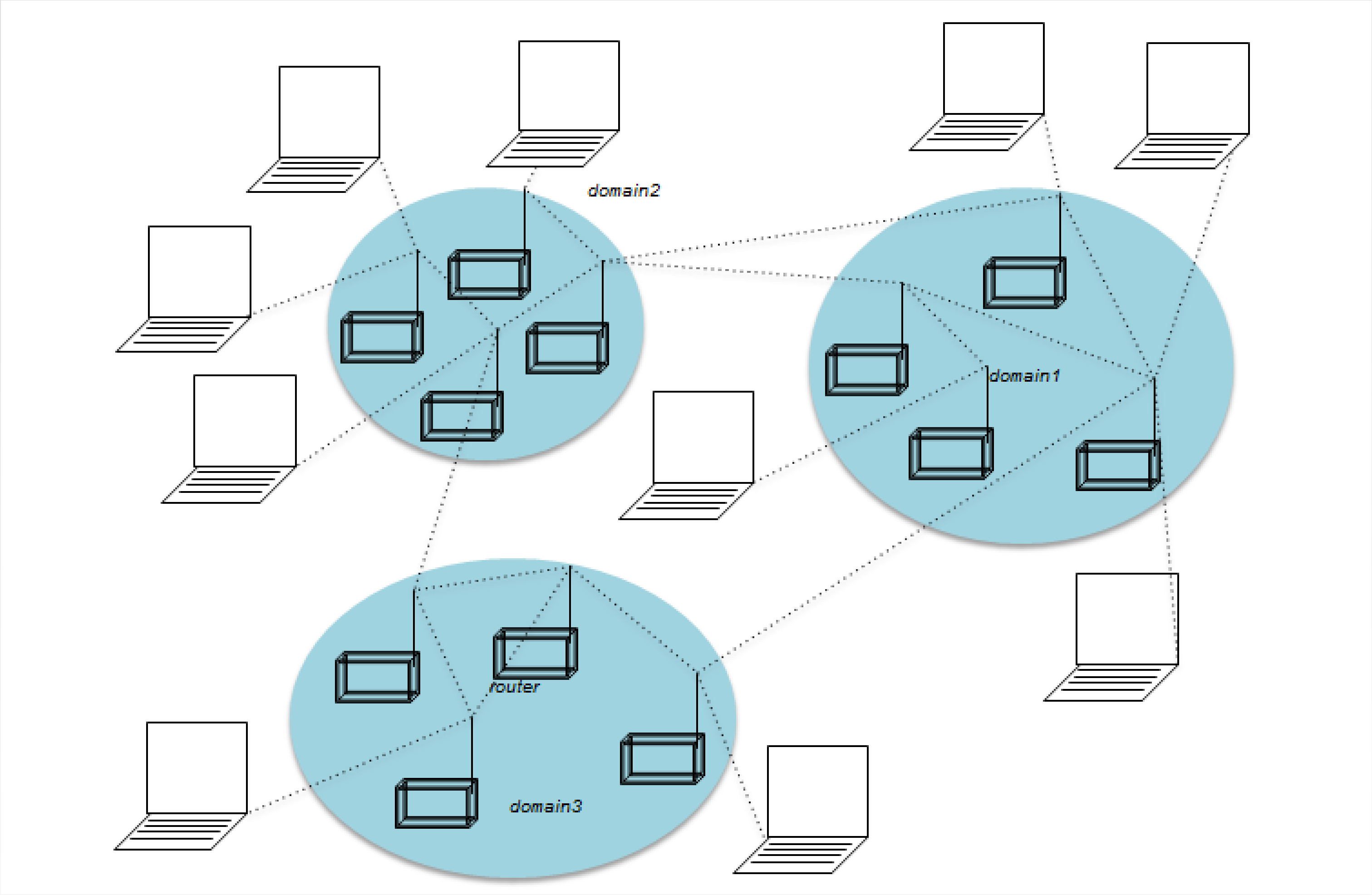
COMP9313: Big Data Management

Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 9: Link Analysis - PageRank

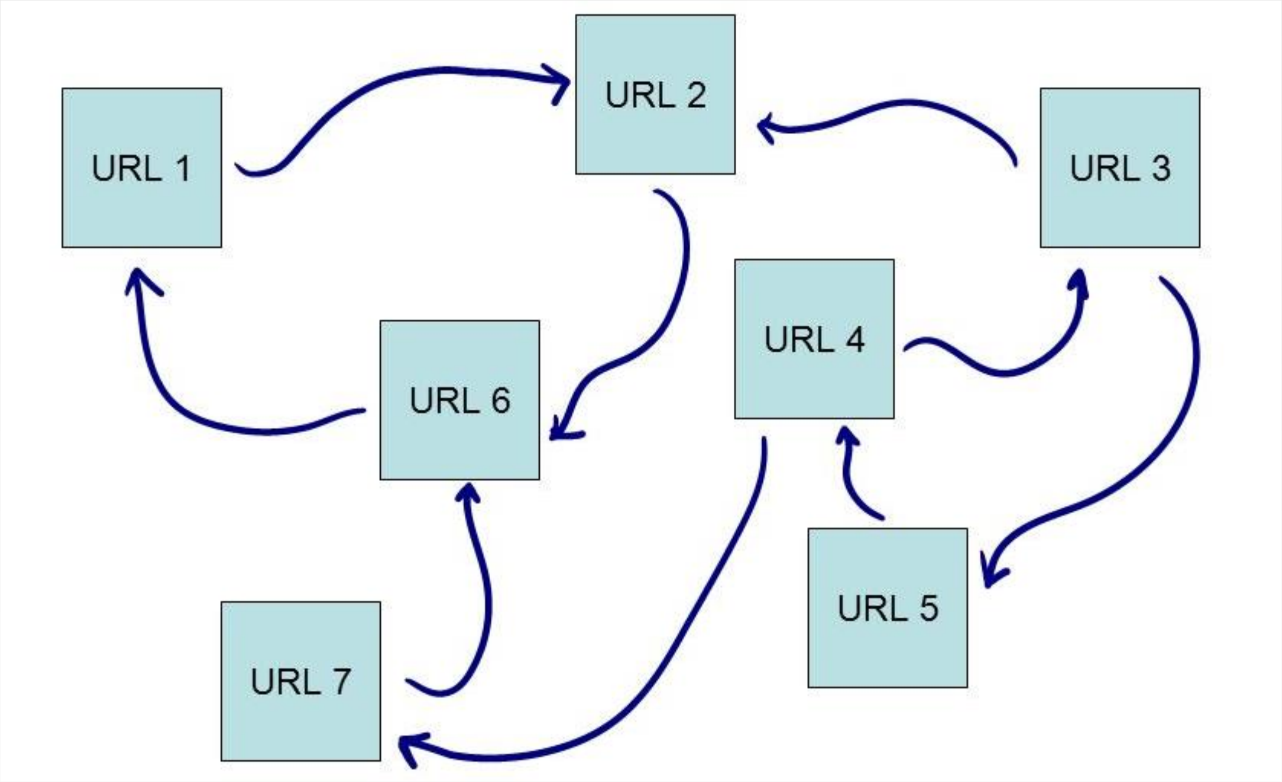
Graph Data: Communication Nets



Internet

Web as a Directed Graph

- Web as a directed graph:
  - Nodes: Webpages
  - Edges: Hyperlinks



Broad Question

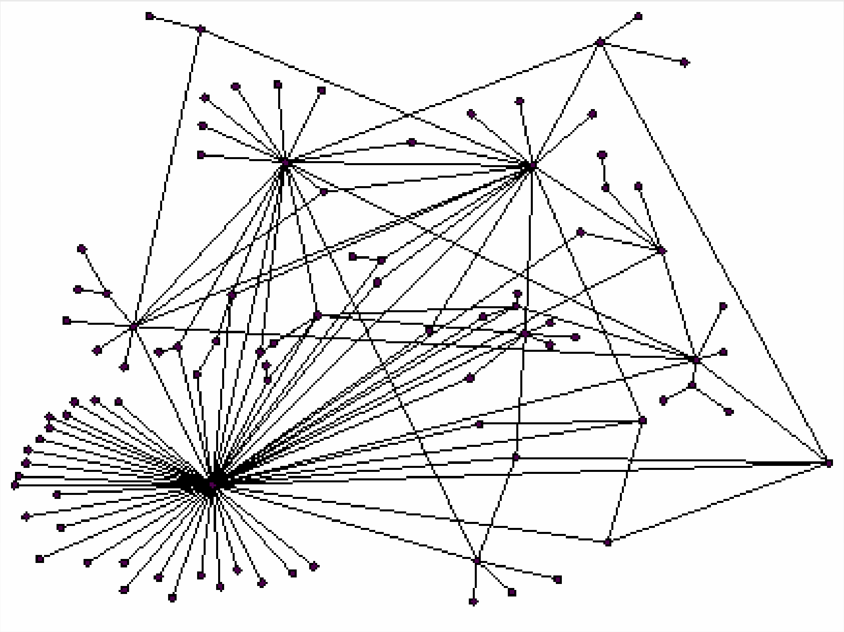
- How to organize the Web?
- First try: Human curated Web directories
  - Yahoo, LookSmart, etc.
- Second try: Web Search
  - Information Retrieval investigates: Find relevant docs in a small and trusted set
    - Newspaper articles, Patents, etc.
  - But: Web is huge, full of untrusted documents, random things, web spam, etc.

- What is the “best” answer to query “newspaper”?
  - No single right answer

Ranking Nodes on the Graph

- All web pages are not equally "important"
  - http://xxx.github.io/ vs. http://www.unsw.edu.au/

- There is large diversity in the web-graph node connectivity. Let's rank the pages by the link structure!



Link Analysis Algorithms

- We will cover the following Link Analysis approaches for computing importance of nodes in a graph:
  - Page Rank
  - Topic-Specific (Personalized) Page Rank
  - HITS
  - … ...

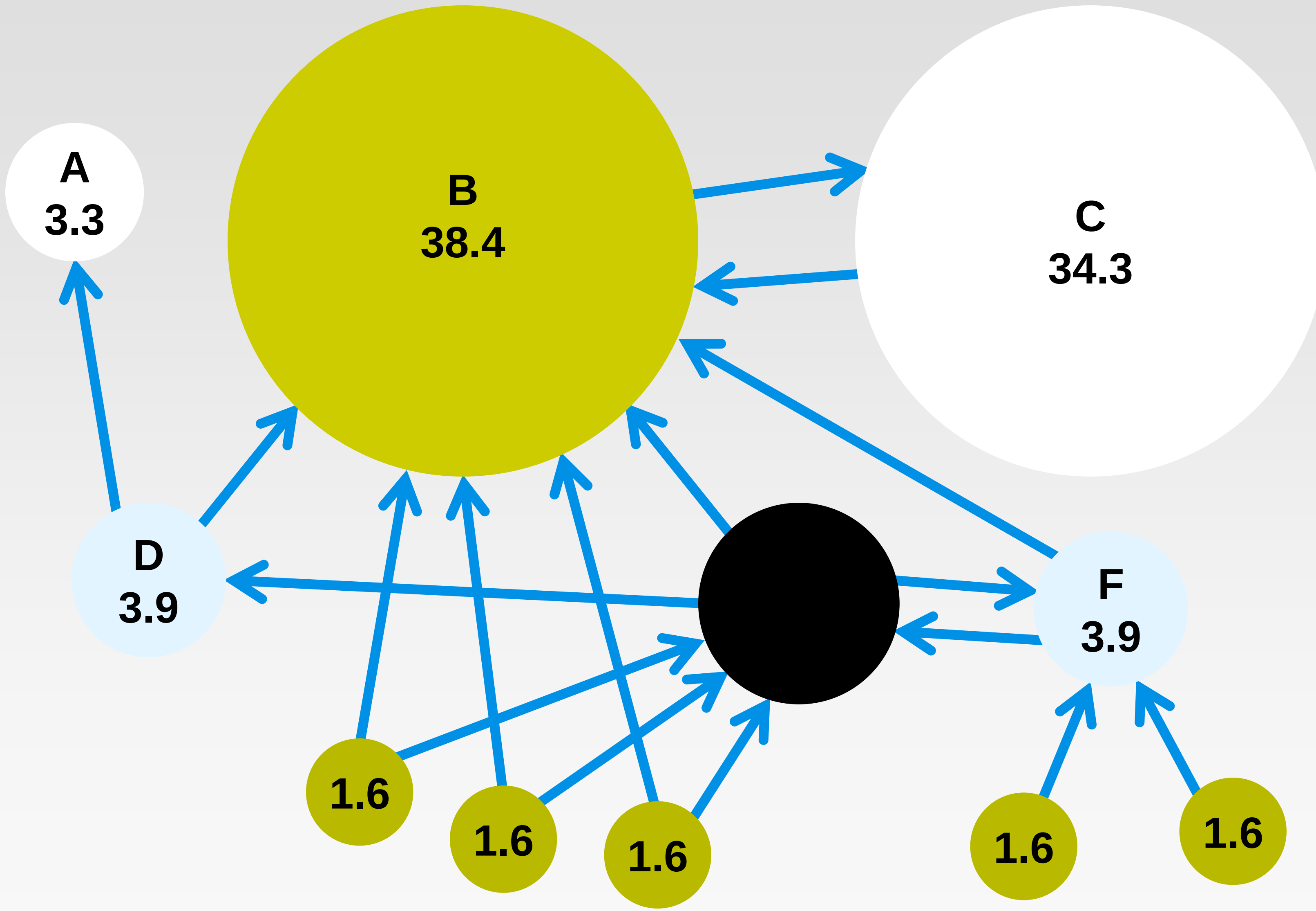
Links as Votes

- Idea: Links as votes
  - Page is more important if it has more links
  - In-coming links? Out-going links?

- Think of in-links as votes:
  - http://www.unsw.edu.au/ has 23,400 in-links
  - http://xxx.github.io/ has 1 in-link

- Are all in-links equal?
  - Links from important pages count more
  - Recursive question!

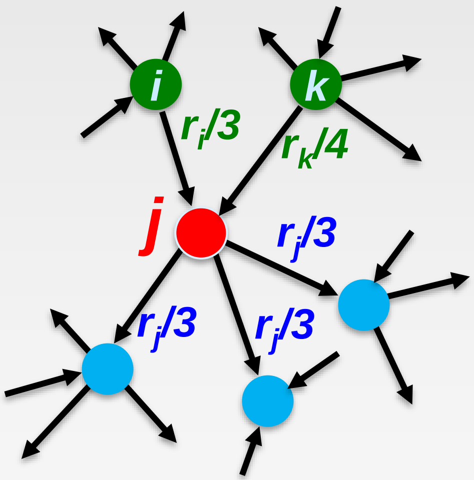
Example: PageRank Scores



Simple Recursive Formulation

- Each link's vote is proportional to the **importance** of its source page
- If page ***j*** with importance r_j has ***n*** out-links, each link gets r_j / n votes
- Page ***j***'s own importance is the sum of the votes on its in-links

$$r_j = r_i/3 + r_k/4$$

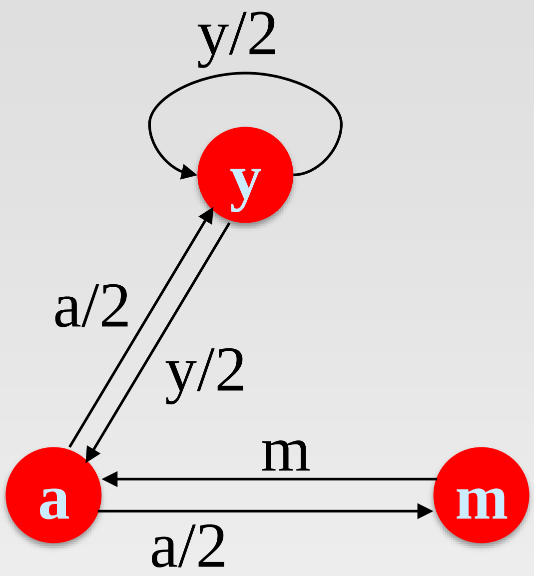


PageRank: The "Flow" Model

- A "vote" from an important page is worth more
- A page is important if it is pointed to by other important pages
- Define a "rank" r_j for page j

$$r_j = \sum_{i \to j} \frac{r_i}{\mathrm{d}_\mathrm{i}}$$

d_i* … out-degree of node *i



"Flow" equations:

$$\mathbf{r_y = r_y/2 + r_a/2}$$
$$\mathbf{r_a = r_y/2 + r_m}$$
$$\mathbf{r_m = r_a/2}$$

Solving the Flow Equations

Flow equations:

$$r_y = r_y/2 + r_a/2$$
$$r_a = r_y/2 + r_m$$
$$r_m = r_a/2$$

- 3 equations, 3 unknowns, no constants
  - No unique solution
  - All solutions equivalent modulo the scale factor
- Additional constraint forces uniqueness:
  - $\boldsymbol{r_y + r_a + r_m = 1}$
  - **Solution:** $\boldsymbol{r_y = \frac{2}{5},\ r_a = \frac{2}{5},\ r_m = \frac{1}{5}}$
- Gaussian elimination method works for small examples, but we need a better method for large web-size graphs
- We need a new formulation!

PageRank: Matrix Formulation

- Stochastic adjacency matrix $\boldsymbol{M}$
  - Let page i has d_i out-links
  - If $i \rightarrow j$, then $M_{ji} = \frac{1}{d_i}$ else $M_{ji} = 0$
    - $\boldsymbol{M}$ is a **column stochastic matrix**
      - Columns sum to 1
- **Rank vector $\boldsymbol{r}$:** vector with an entry per page
  - r_i is the importance score of page i
  - $\sum_i r_i = 1$
- The flow equations can be written

$$\boldsymbol{r} = \boldsymbol{M} \cdot \boldsymbol{r}$$

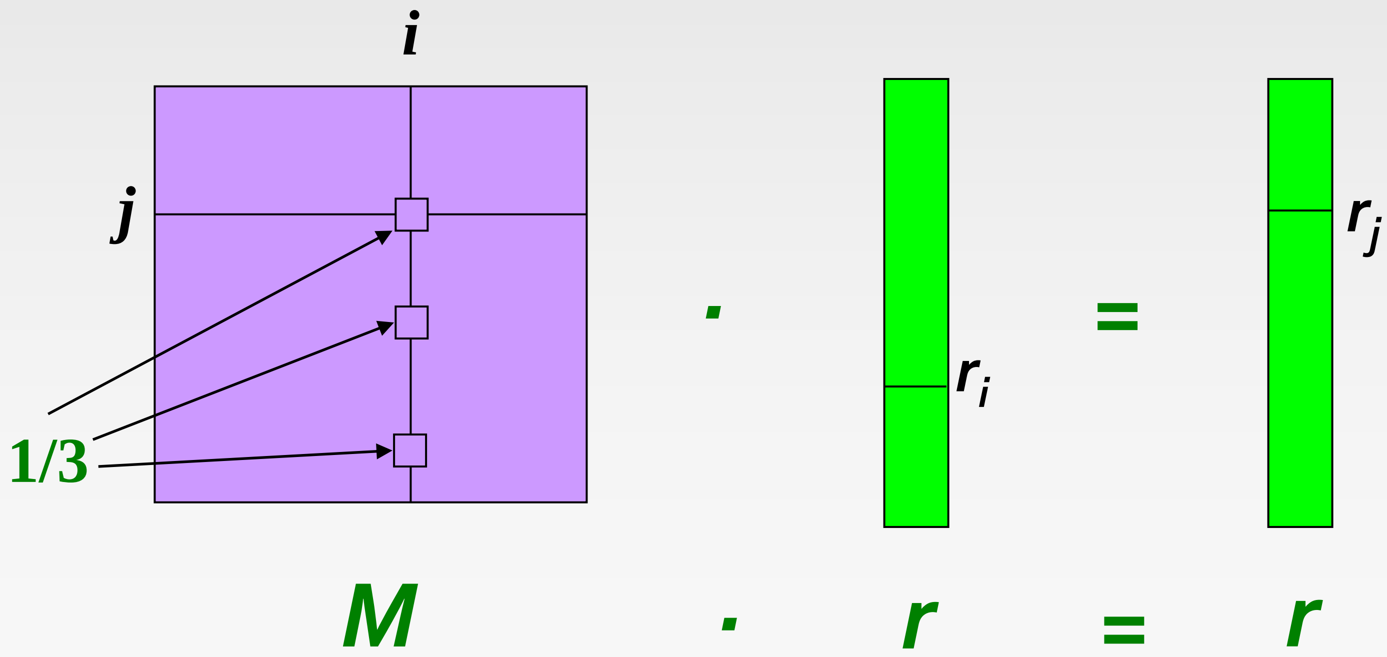
Example

- Remember the flow equation: $$r_j = \sum_{i \to j} \frac{r_i}{\mathrm{d_i}}$$

- Flow equation in the matrix form

$$M \cdot r = r$$

  - Suppose page *i* links to 3 pages, including *j*



Eigenvector Formulation

- The flow equations can be written

$$\boldsymbol{r} = \boldsymbol{M} \cdot \boldsymbol{r}$$

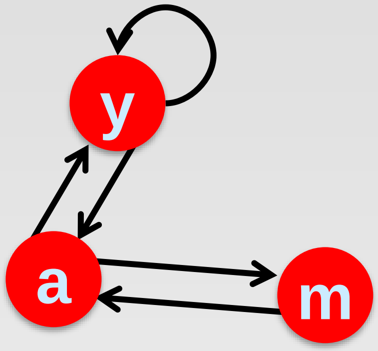
- So the **rank vector** ***r*** is an **eigenvector** of the stochastic web matrix ***M***
  - In fact, its first or principal eigenvector, with corresponding eigenvalue ***1***
    - Largest eigenvalue of ***M*** is **1** since ***M*** is column stochastic (with non-negative entries)
      - *We know* ***r*** *is unit length and each column of* ***M*** *sums to one, so* $\boldsymbol{Mr} \leq \mathbf{1}$

NOTE: ***x*** is an eigenvector with the corresponding eigenvalue **λ** if:

$$\boldsymbol{Ax} = \boldsymbol{\lambda x}$$

- We can now efficiently solve for r!
  - The method is called Power iteration

Example: Flow Equations & M



	y	a	m
y	½	½	0
a	½	0	1
m	0	½	0

$$\boldsymbol{r = M \cdot r}$$

$$r_y = r_y/2 + r_a/2$$
$$r_a = r_y/2 + r_m$$
$$r_m = r_a/2$$

$$\begin{bmatrix} y \\ a \\ m \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} y \\ a \\ m \end{bmatrix}$$

Power Iteration Method

- Given a web graph with n nodes, where the nodes are pages and edges are hyperlinks
- **Power iteration:** a simple iterative scheme
  - Suppose there are *N* web pages
  - Initialize: $\mathbf{r}^{(0)} = [1/N,\ldots.,1/N]^T$
  - Iterate: $\mathbf{r}^{(t+1)} = \mathbf{M} \cdot \mathbf{r}^{(t)}$
  - Stop when $|\mathbf{r}^{(t+1)} - \mathbf{r}^{(t)}|_1 < \varepsilon$

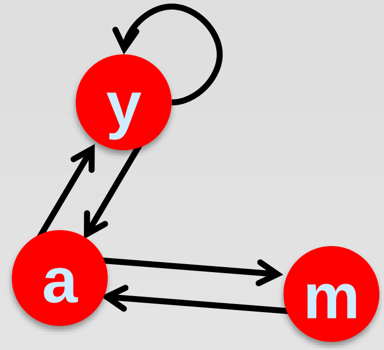
$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{\mathrm{d}_i}$$

d_i …. out-degree of node i

$|\mathbf{x}|_1 = \sum_{1 \le i \le N} |x_i|$ is the $\mathbf{L_1}$ norm
Can use any other vector norm, e.g., Euclidean

PageRank: How to solve?

- **Power Iteration:**
  - Set $r_j = 1/N$
  - **1:** $r'_j = \sum_{i \to j} \frac{r_i}{d_i}$
  - **2:** $r = r'$
  - Goto **1**



	y	a	m
y	½	½	0
a	½	0	1
m	0	½	0

$$\mathbf{r_y = r_y/2 + r_a/2}$$
$$\mathbf{r_a = r_y/2 + r_m}$$
$$\mathbf{r_m = r_a/2}$$

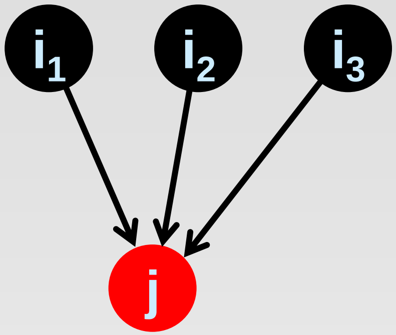
- **Example:**

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{matrix} 1/3 \\ 1/3 \\ 1/3 \end{matrix} \qquad \ldots \qquad \begin{matrix} 6/15 \\ 6/15 \\ 3/15 \end{matrix}$$

Iteration 0, 1, 2, …

Random Walk Interpretation

- **Imagine a random web surfer:**
  - At any time $\boldsymbol{t}$, surfer is on some page $\boldsymbol{i}$
  - At time $\boldsymbol{t+1}$, the surfer follows an out-link from $\boldsymbol{i}$ uniformly at random
  - Ends up on some page $\boldsymbol{j}$ linked from $\boldsymbol{i}$
  - Process repeats indefinitely
- **Let:**
  - $\boldsymbol{p(t)}$ … vector whose $\boldsymbol{i}^{\text{th}}$ coordinate is the prob. that the surfer is at page $\boldsymbol{i}$ at time $\boldsymbol{t}$
  - So, $\boldsymbol{p(t)}$ is a probability distribution over pages



$$r_j = \sum_{i \to j} \frac{r_i}{\mathrm{d}_{\mathrm{out}}(\mathrm{i})}$$

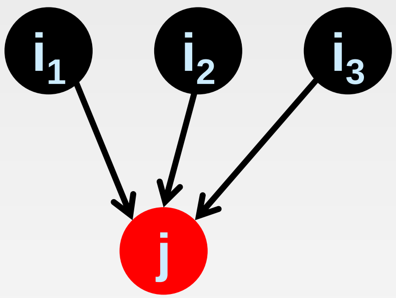
The Stationary Distribution

- **Where is the surfer at time *t+1*?**
  - Follows a link uniformly at random

    $$\boldsymbol{p}(\boldsymbol{t}+\mathbf{1}) = \boldsymbol{M} \cdot \boldsymbol{p}(\boldsymbol{t})$$

- Suppose the random walk reaches a state $\boldsymbol{p}(\boldsymbol{t}+\mathbf{1}) = \boldsymbol{M} \cdot \boldsymbol{p}(\boldsymbol{t}) = \boldsymbol{p}(\boldsymbol{t})$

  then $\boldsymbol{p}(\boldsymbol{t})$ is **stationary distribution** of a random walk
- **Our original rank vector** $\boldsymbol{r}$ satisfies $\boldsymbol{r} = \boldsymbol{M} \cdot \boldsymbol{r}$
  - **So, $\boldsymbol{r}$ is a stationary distribution for the random walk**



$$p(t+1) = \mathrm{M} \cdot p(t)$$

Existence and Uniqueness

- A central result from the theory of random walks (a.k.a. Markov processes):

For graphs that satisfy **certain conditions**, the **stationary distribution is unique** and eventually will be reached no matter what the initial probability distribution at time **t = 0**

PageRank: Two Questions

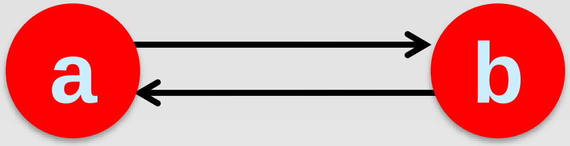
$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{\mathrm{d_i}}$$

or
equivalently

$$r = Mr$$

- Does this converge?
- Does it converge to what we want?

Does this converge?



$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{\mathrm{d_i}}$$

- Example:

r_a	1	0	1	0	…
r_b	0	1	0	1	…

Iteration 0, 1, 2, …

Does it converge to what we want?

$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{\mathrm{d_i}}$$

- Example:

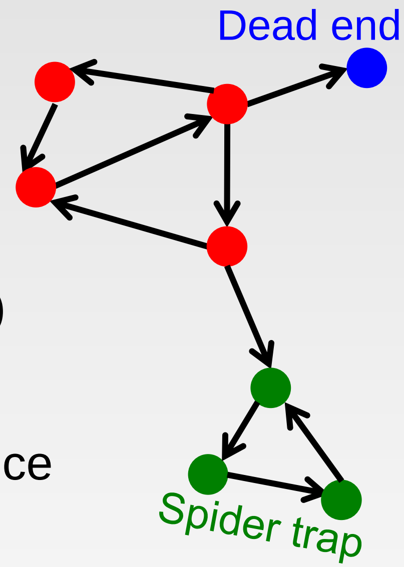
r_a	1	0	0	0
r_b	0	1	0	0

Iteration 0, 1, 2, …

PageRank: Problems

2 problems:

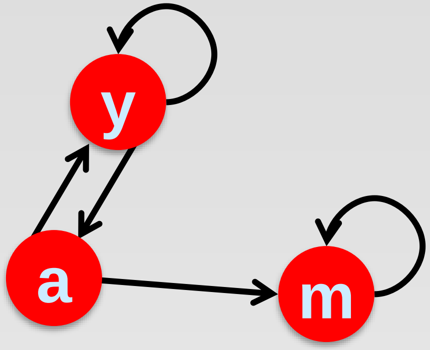
- **(1)** Some pages are **dead ends** (have no out-links)
  - Random walk has "nowhere" to go to
  - Such pages cause importance to "leak out"



- **(2)** **Spider traps:** (all out-links are within the group)
  - Random walked gets "stuck" in a trap
  - And eventually spider traps absorb all importance

Problem: Spider Traps

- **Power Iteration:**
  - Set $r_j = 1$
  - $r_j = \sum_{i \to j} \frac{r_i}{d_i}$
    - And iterate



m is a spider trap

	y	a	m
y	½	½	0
a	½	0	0
m	0	½	1

$r_y = r_y/2 + r_a/2$

$r_a = r_y/2$

$r_m = r_a/2 + r_m$

- **Example:**

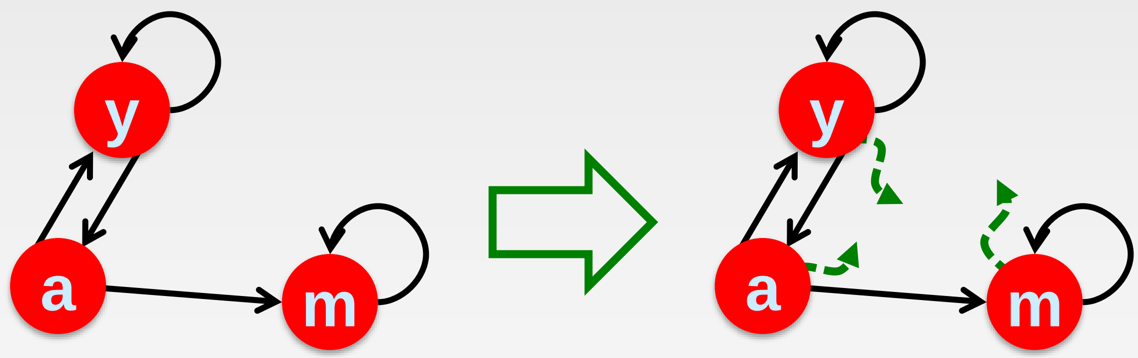
$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{matrix} 1/3 & 2/6 & 3/12 & 5/24 & & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \dots & 0 \\ 1/3 & 3/6 & 7/12 & 16/24 & & 1 \end{matrix}$$

Iteration 0, 1, 2, …

All the PageRank score gets "trapped" in node m.

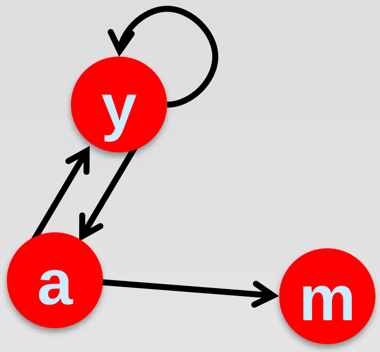
Solution: Teleport!

- The Google solution for spider traps: **At each time step, the random surfer has two options**
  - With prob. β, follow a link at random
  - With prob. **1-**β, jump to some random page
  - Common values for β are in the range 0.8 to 0.9
- Surfer will teleport out of spider trap within a few time steps



Problem: Dead Ends

- **Power Iteration:**
  - Set $r_j = 1$
  - $r_j = \sum_{i \to j} \frac{r_i}{d_i}$
    - And iterate



	y	a	m
y	½	½	0
a	½	0	0
m	0	½	0

$\mathbf{r_y = r_y/2 + r_a/2}$

$\mathbf{r_a = r_y/2}$

$\mathbf{r_m = r_a/2}$

- **Example:**

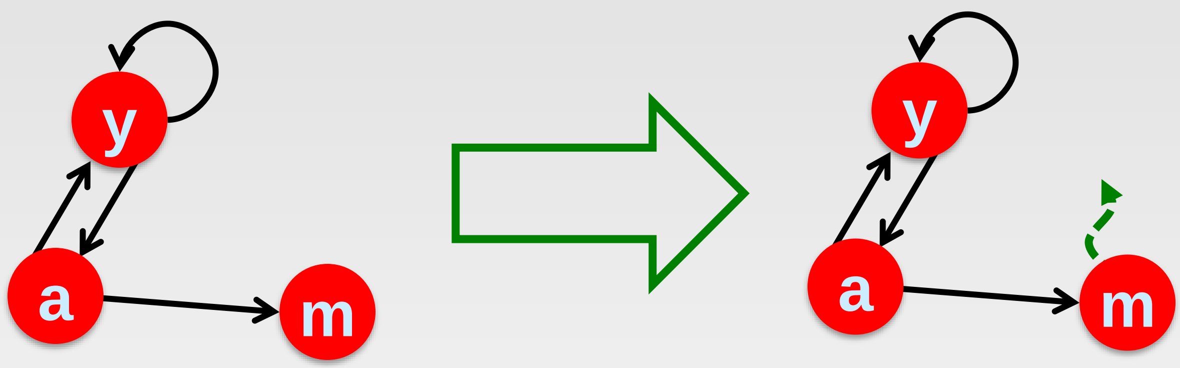
$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{matrix} 1/3 & 2/6 & 3/12 & 5/24 & & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \ldots & 0 \\ 1/3 & 1/6 & 1/12 & 2/24 & & 0 \end{matrix}$$

Iteration 0, 1, 2, …

Here the PageRank “leaks” out since the matrix is not stochastic.

Solution: Always Teleport!

- Teleports: Follow random teleport links with probability 1.0 from dead-ends
  - Adjust matrix accordingly



	y	a	m
y	½	½	0
a	½	0	0
m	0	½	0

	y	a	m
y	½	½	⅓
a	½	0	⅓
m	0	½	⅓

Why Teleports Solve the Problem?

Why are dead-ends and spider traps a problem and why do teleports solve the problem?

- **Spider-traps** are not a problem, but with traps PageRank scores are **not** what we want
  - **Solution:** Never get stuck in a spider trap by teleporting out of it in a finite number of steps
- **Dead-ends** are a problem
  - The matrix is not column stochastic so our initial assumptions are not met
  - **Solution:** Make matrix column stochastic by always teleporting when there is nowhere else to go

Google's Solution: Random Teleports

- **Google's solution that does it all:**
  At each step, random surfer has two options:
  - With probability β, follow a link at random
  - With probability ***1-β***, jump to some random page

- **PageRank equation** [Brin-Page, 98]

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$$

d_i … out-degree of node i

This formulation assumes that ***M*** has no dead ends. We can either preprocess matrix ***M*** to remove all dead ends or explicitly follow random teleport links with probability 1.0 from dead-ends.

The Google Matrix

- **PageRank equation** [Brin-Page, '98]

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$$

- **The Google Matrix *A*:**

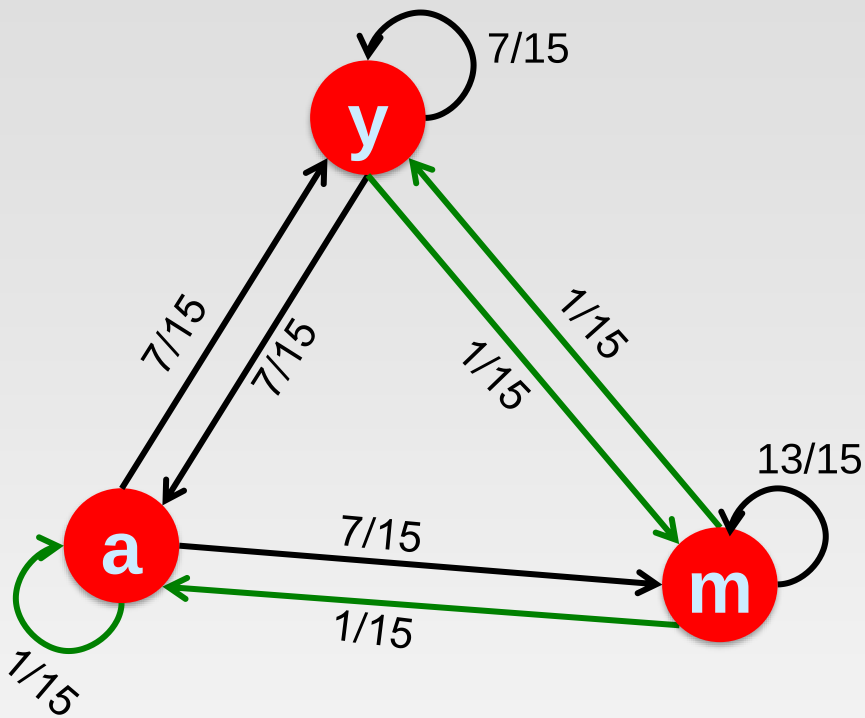
$$A = \beta\, M + (1-\beta)\left[\frac{1}{N}\right]_{N \times N}$$

$[1/N]_{NxN}$…N by N matrix where all entries are 1/N

- **We have a recursive problem:** $\boldsymbol{r} = \boldsymbol{A} \cdot \boldsymbol{r}$
  And the Power method still works!

- **What is β?**
  - In practice $\beta = 0.8, 0.9$ (make 5 steps on avg., jump)

Random Teleports ($\beta = 0.8$)



$$0.8 \overset{\textbf{M}}{\boxed{\begin{matrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{matrix}}} + 0.2 \overset{[1/N]_{N\times N}}{\boxed{\begin{matrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{matrix}}}$$

$$\begin{array}{c} y \\ a \\ m \end{array} \underset{\textbf{A}}{\boxed{\begin{matrix} 7/15 & 7/15 & 1/15 \\ 7/15 & 1/15 & 1/15 \\ 1/15 & 7/15 & 13/15 \end{matrix}}}$$

$$\begin{matrix} y \\ a \\ m \end{matrix} \; = \; \begin{matrix} 1/3 \\ 1/3 \\ 1/3 \end{matrix} \quad \begin{matrix} 0.33 \\ 0.20 \\ 0.46 \end{matrix} \quad \begin{matrix} 0.24 \\ 0.20 \\ 0.52 \end{matrix} \quad \begin{matrix} 0.26 \\ 0.18 \\ 0.56 \end{matrix} \quad \dots \quad \begin{matrix} 7/33 \\ 5/33 \\ 21/33 \end{matrix}$$

Computing Page Rank

- Key step is matrix-vector multiplication
  - $\boldsymbol{r}^{new} = \boldsymbol{A} \cdot \boldsymbol{r}^{old}$
- Easy if we have enough main memory to hold **A**, $\mathbf{r}^{old}$, $\mathbf{r}^{new}$
- Say N = 1 billion pages
  - We need 4 bytes for each entry (say)
  - 2 billion entries for vectors, approx 8GB
  - Matrix A has N^2 entries
    - 10^{18} is a large number!

$$\mathbf{A} = \beta \cdot \mathbf{M} + (1-\beta)\ [1/N]_{N \times N}$$

$$A = 0.8 \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} + 0.2 \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}$$

$$= \begin{bmatrix} 7/15 & 7/15 & 1/15 \\ 7/15 & 1/15 & 1/15 \\ 1/15 & 7/15 & 13/15 \end{bmatrix}$$

Matrix Formulation

- Suppose there are ***N*** pages
- Consider page ***i***, with $\mathbf{d}_i$ out-links
- We have $M_{ji} = 1/|d_i|$ when $i \rightarrow j$
  and $M_{ji} = 0$ otherwise
- The random teleport is equivalent to:
  - Adding a **teleport link** from ***i*** to every other page and setting transition probability to $(1-\beta)/N$
  - Reducing the probability of following each out-link from $1/|d_i|$ to $\beta/|d_i|$
  - **Equivalent:** Tax each page a fraction $(1-\beta)$ of its score and redistribute evenly

Rearranging the Equation

- $\boldsymbol{r} = \boldsymbol{A} \cdot \boldsymbol{r}$, **where** $\boldsymbol{A_{ji}} = \boldsymbol{\beta}\, \boldsymbol{M_{ji}} + \frac{\boldsymbol{1-\beta}}{\boldsymbol{N}}$
- $r_j = \sum_{\mathrm{i}=1}^{N} A_{ji} \cdot r_i$
- $r_j = \sum_{i=1}^{N} \left[\beta\, M_{ji} + \frac{1-\beta}{N}\right] \cdot r_i$

$$= \sum_{\mathrm{i}=1}^{N} \beta\, M_{ji} \cdot r_i + \frac{1-\beta}{N} \sum_{\mathrm{i}=1}^{N} r_i$$

$$= \sum_{\mathrm{i}=1}^{N} \beta\, M_{ji} \cdot r_i + \frac{1-\beta}{N}$$

since $\sum r_i = 1$

- **So we get:** $\boldsymbol{r} = \boldsymbol{\beta}\, \boldsymbol{M} \cdot \boldsymbol{r} + \left[\frac{\boldsymbol{1-\beta}}{\boldsymbol{N}}\right]_{\boldsymbol{N}}$

Note: Here we assumed **M** has no dead-ends

$[x]_N$ … a vector of length N with all entries x

Sparse Matrix Formulation

- We just rearranged the **PageRank equation**

$$\boldsymbol{r} = \boldsymbol{\beta M} \cdot \boldsymbol{r} + \left[\frac{\mathbf{1} - \boldsymbol{\beta}}{\boldsymbol{N}}\right]_{\boldsymbol{N}}$$

  - where $\mathbf{[(1-\beta)/N]_N}$ is a vector with all ***N*** entries $\mathbf{(1-\beta)/N}$

- ***M*** is a **sparse matrix!** (with no dead-ends)
  - 10 links per node, approx 10N entries
- So in each iteration, we need to:
  - Compute $\boldsymbol{r}^{\text{new}} = \beta\, \boldsymbol{M} \cdot \boldsymbol{r}^{\text{old}}$
  - Add a constant value $\mathbf{(1-\beta)/N}$ to each entry in $\boldsymbol{r}^{\text{new}}$
    - **Note if M contains dead-ends then $\sum_j r_j^{new} < 1$ and we also have to renormalize $\boldsymbol{r}^{\text{new}}$ so that it sums to 1**

PageRank: The Complete Algorithm

- **Input:** Graph $\boldsymbol{G}$ and parameter $\boldsymbol{\beta}$
  - Directed graph $\boldsymbol{G}$ (can have **spider traps** and **dead ends**)
  - Parameter $\boldsymbol{\beta}$
- **Output:** PageRank vector $\boldsymbol{r}^{\boldsymbol{new}}$
  - **Set:** $r_j^{old} = \frac{1}{N}$
  - **repeat until convergence:** $\sum_j \left| r_j^{new} - r_j^{old} \right| > \varepsilon$
    - $\forall j$: $\boldsymbol{r'}_{\boldsymbol{j}}^{\boldsymbol{new}} = \sum_{\boldsymbol{i \to j}} \boldsymbol{\beta} \frac{\boldsymbol{r_i^{old}}}{\boldsymbol{d_i}}$

      $\boldsymbol{r'}_{\boldsymbol{j}}^{\boldsymbol{new}} = \boldsymbol{0}$ if in-degree of $\boldsymbol{j}$ is **0**
    - **Now re-insert the leaked PageRank:**

      $\forall \boldsymbol{j}$: $\boldsymbol{r}_{\boldsymbol{j}}^{\boldsymbol{new}} = \boldsymbol{r'}_{\boldsymbol{j}}^{\boldsymbol{new}} + \frac{\boldsymbol{1-S}}{\boldsymbol{N}}$ **where:** $S = \sum_j r'^{new}_j$
    - $\boldsymbol{r}^{\boldsymbol{old}} = \boldsymbol{r}^{\boldsymbol{new}}$

If the graph has no dead-ends then the amount of leaked PageRank is **1-β**. But since we have dead-ends the amount of leaked PageRank may be larger. We have to explicitly account for it by computing **S**.

Sparse Matrix Encoding

- Encode sparse matrix using only nonzero entries
  - Space proportional roughly to number of links
  - Say 10N, or 4*10*1 billion = 40GB
  - **Still won't fit in memory, but will fit on disk**

source node	degree	destination nodes
0	3	1, 5, 7
1	5	17, 64, 113, 117, 245
2	2	13, 23

Basic Algorithm: Update Step

- Assume enough RAM to fit r^{new} into memory
  - Store r^{old} and matrix **M** on disk
- 1 step of power-iteration is:

> **Initialize** all entries of $\mathbf{r^{new}} = \mathbf{(1-\beta) / N}$
> For each page $\boldsymbol{i}$ (of out-degree $\boldsymbol{d_i}$):
>   Read into memory: $\boldsymbol{i, d_i, dest_1, \ldots, dest_{d_i}, r^{old}(i)}$
>   For $\mathbf{j = 1 \ldots d_i}$
> $\mathbf{r^{new}(dest_j) \mathrel{+}= \beta\, r^{old}(i) / \boldsymbol{d_i}}$

r^{new}: 0, 1, 2, 3, 4, 5

source	degree	destination
0	3	1, 5, 6
1	4	17, 64, 113, 117
2	2	13, 23

r^{old}: 0, 1, 2, 3, 4, 5, 6

Analysis

- Assume enough RAM to fit r^{new} into memory
  - Store $\boldsymbol{r}^{old}$ and matrix $\boldsymbol{M}$ on disk
- In each iteration, we have to:
  - Read $\boldsymbol{r}^{old}$ and $\boldsymbol{M}$
  - Write $\boldsymbol{r}^{new}$ back to disk
  - **Cost per iteration of Power method:**
    = $2|\boldsymbol{r}| + |\boldsymbol{M}|$
- Question:
  - What if we could not even fit $\boldsymbol{r}^{new}$ in memory?
  - Split $\boldsymbol{r}^{new}$ into blocks. Details ignored

Some Problems with Page Rank

- **Measures generic popularity of a page**
  - Biased against topic-specific authorities
  - **Solution:** Topic-Specific (Personalized) PageRank (**next**)
- **Uses a single measure of importance**
  - Other models of importance
  - **Solution:** Hubs-and-Authorities

PageRank in MapReduce

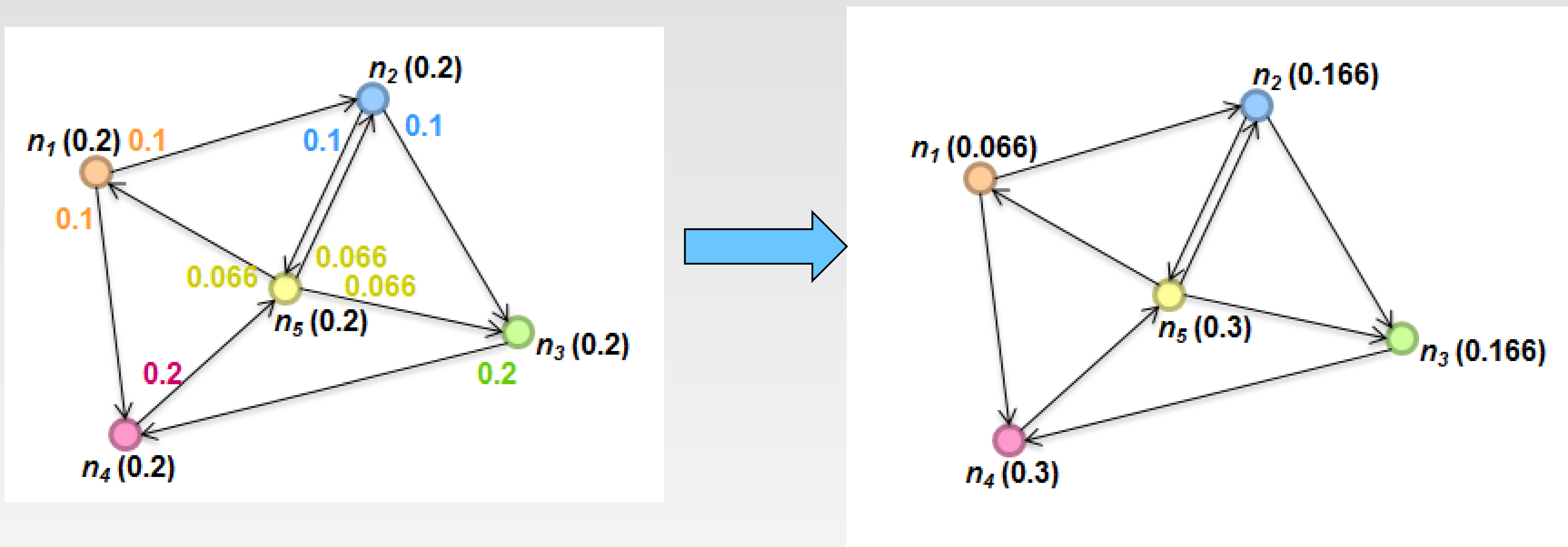
PageRank Computation Review

- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm:
  - Start with seed r_i values
  - Each page distributes r_i "credit" to all pages it links to
  - Each target page t_j adds up "credit" from multiple in-bound links to compute r_j
  - Iterate until values converge

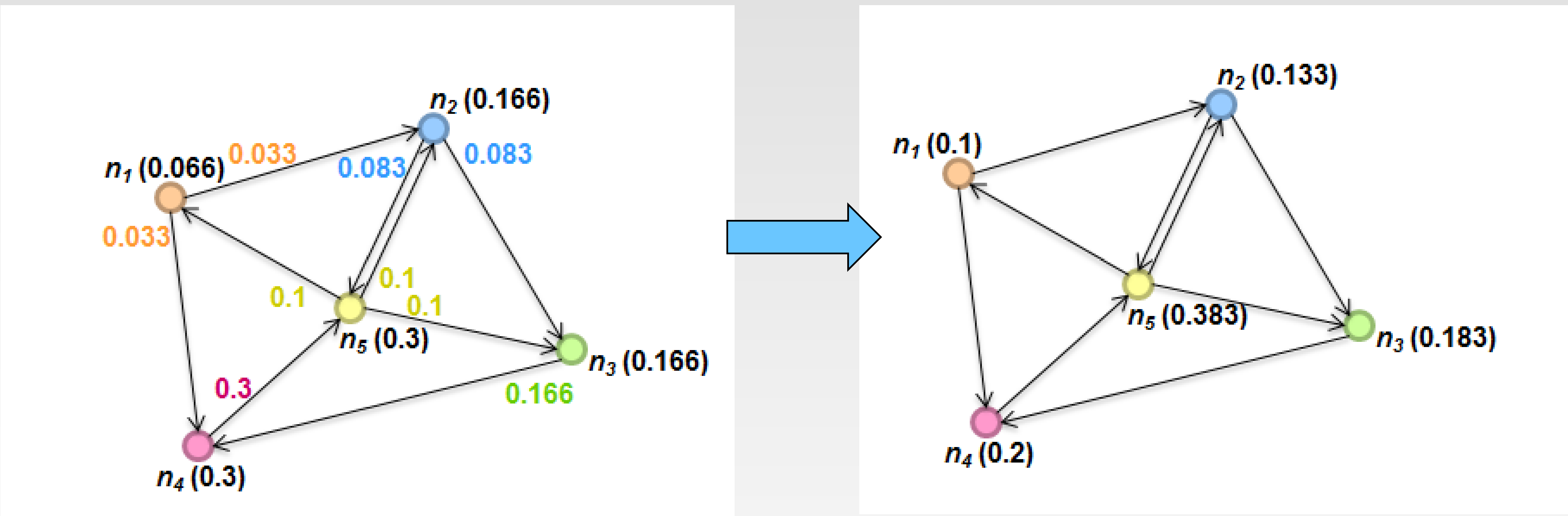
Simplified PageRank

- First, tackle the simple case:
  - No teleport
  - No dead ends
- Then, factor in these complexities…
  - How to deal with the teleport probability?
  - How to deal with dead ends?

Sample PageRank Iteration (1)



Sample PageRank Iteration (2)



PageRank in MapReduce

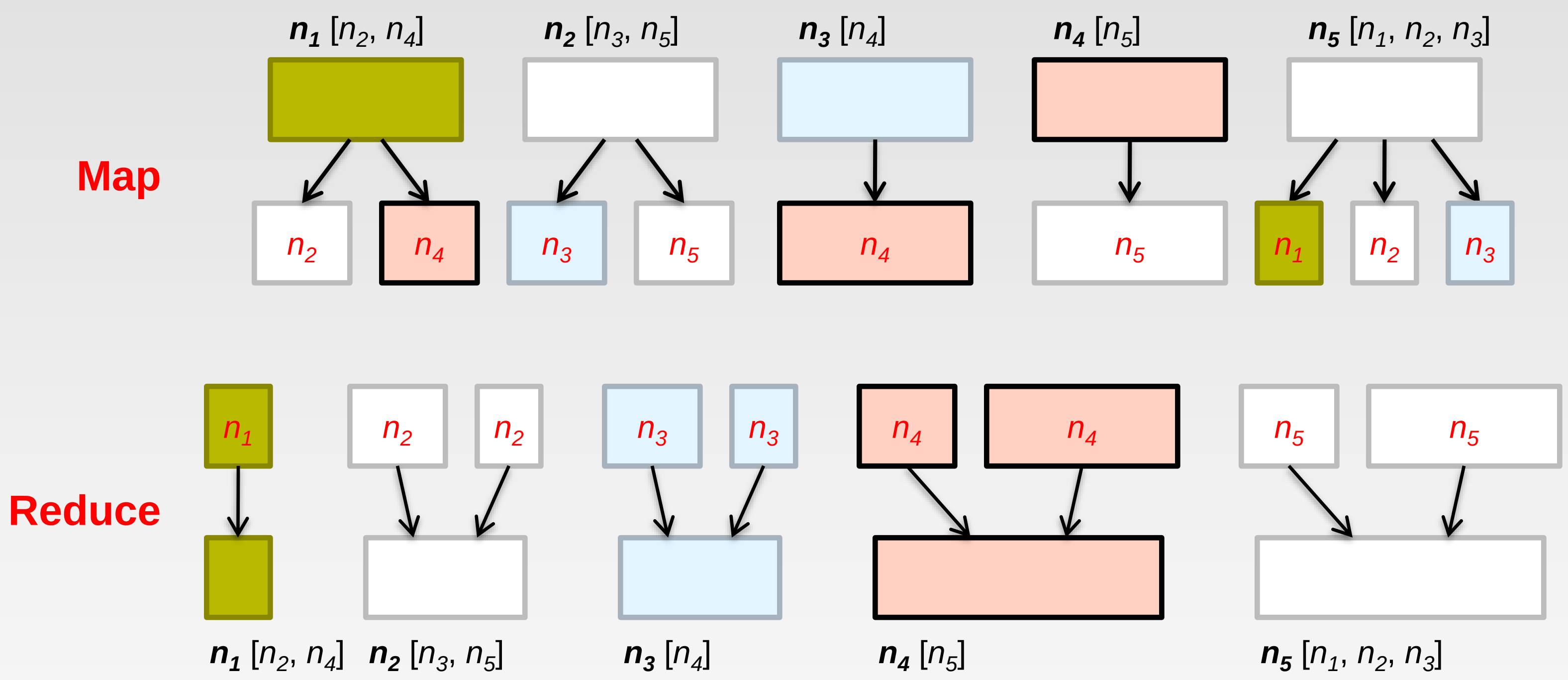
- One iteration of the PageRank algorithm involves taking an estimated PageRank vector r and computing the next estimate r′ by

$$\boldsymbol{r} = \boldsymbol{\beta}\,\boldsymbol{M}\cdot\boldsymbol{r} + \left[\frac{\mathbf{1}-\boldsymbol{\beta}}{\boldsymbol{N}}\right]_{\boldsymbol{N}}$$

- Mapper: input – a line containing node u, r_u, a list of out-going neighbors of u
  - For each neighbor v, emit(v, r_u/deg(u))
  - Emit (u, a list of out-going neighbors of u)

- Reducer: input – (node v, a list of values <r_u/deg(u), …>)
  - Aggregate the results according to the equation to compute r'_v
  - Emit node v, r'_v, a list of out-going neighbors of v

PageRank in MapReduce (One Iteration)



PageRank Pseudo-Code

```
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                               ▷ Sums incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```

Complete PageRank

- Two additional complexities
  - What is the proper treatment of dangling nodes?
  - How do we factor in the random jump factor?
- Solution:
  - If a node's adjacency list is empty, distribute its value to all nodes evenly.
    - In mapper, for such a node *i*, emit (nid m, r_i/N) for each node m in the graph
  - Add the teleport value
    - In reducer, M.PageRank = $\beta * s + (1-\beta) / N$

Graphs and MapReduce

- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: “traversing” the graph
- Generic recipe:
  - Represent graphs as adjacency lists
  - Perform local computations in mapper
  - Pass along partial results via outlinks, keyed by destination node
  - Perform aggregation in reducer on inlinks to a node
  - Iterate until convergence: controlled by external “driver”
  - Don’t forget to pass the graph structure between iterations

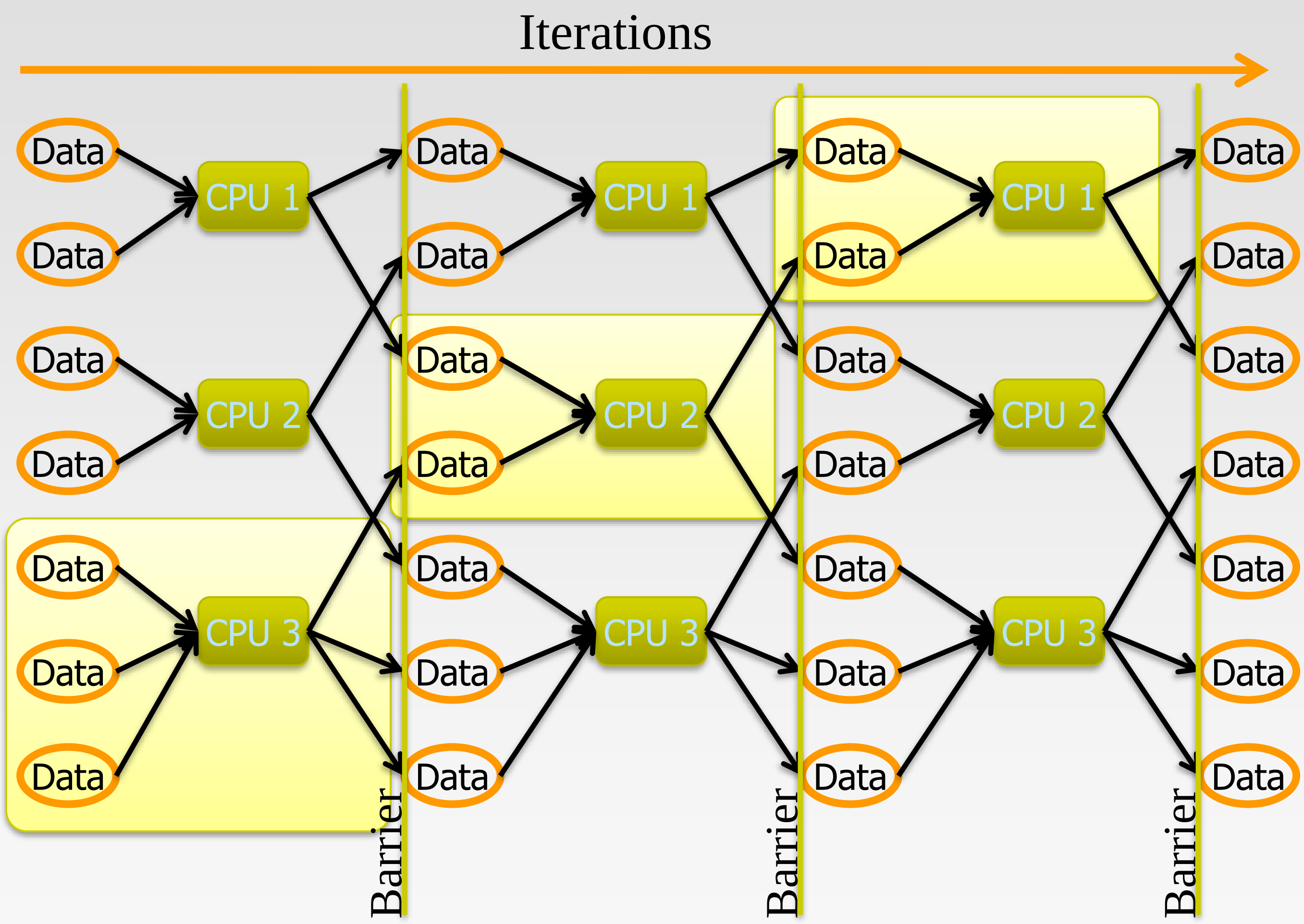
Issues with MapReduce on Graph Processing

- MapReduce Does not support iterative graph computations:
  - External driver. Huge I/O incurs
  - No mechanism to support global data structures that can be accessed and updated by all mappers and reducers
    - Passing information is only possible within the local graph structure – through adjacency list
    - Some “wasted” computations are needed

- MapReduce algorithms are often impractical on large, dense graphs.
  - The amount of intermediate data generated is on the order of the number of edges.
  - For dense graphs, MapReduce running time would be dominated by copying intermediate data across the network.

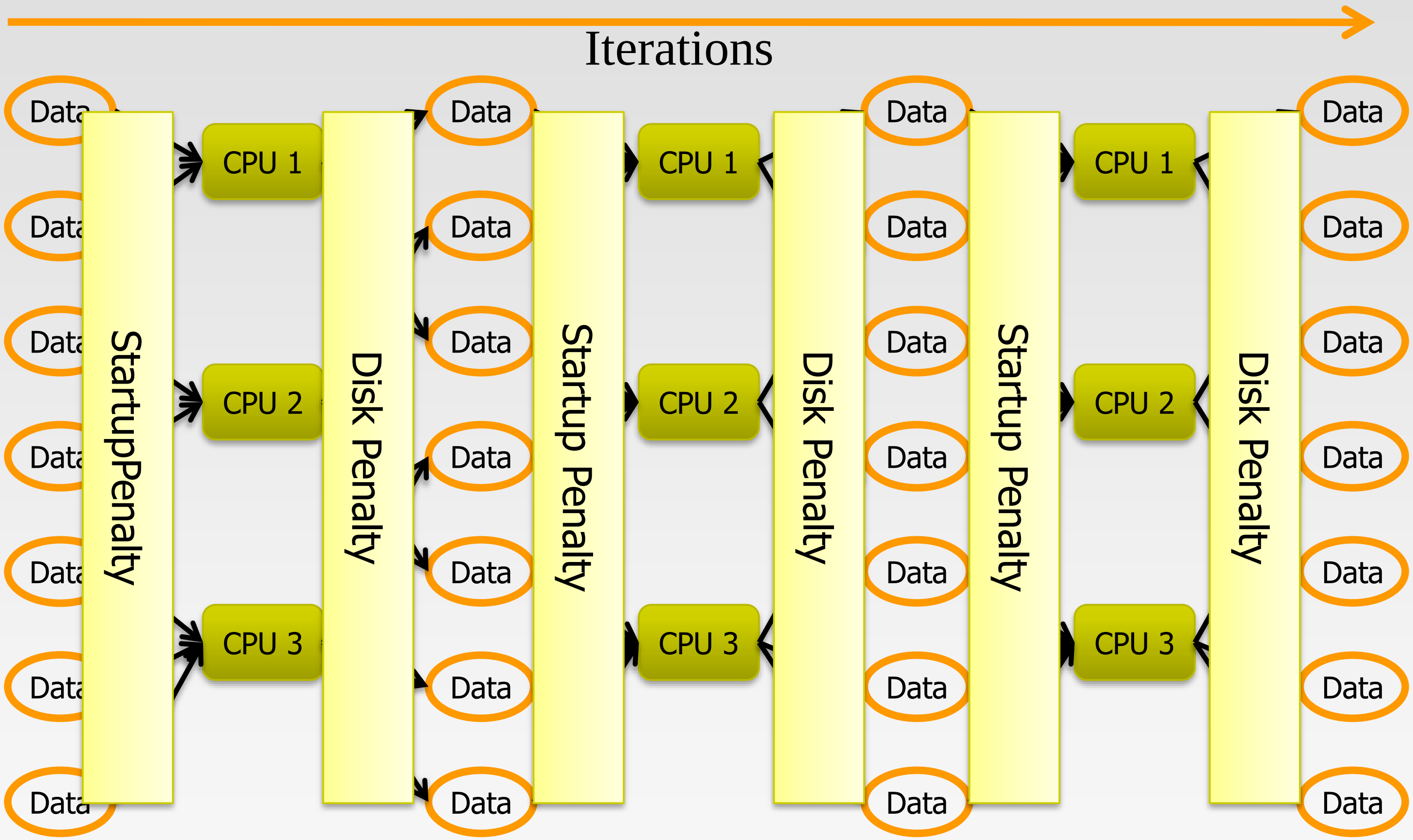
Iterative MapReduce

- Only a subset of data needs computation:



Iterative MapReduce

- System is not optimized for iteration:

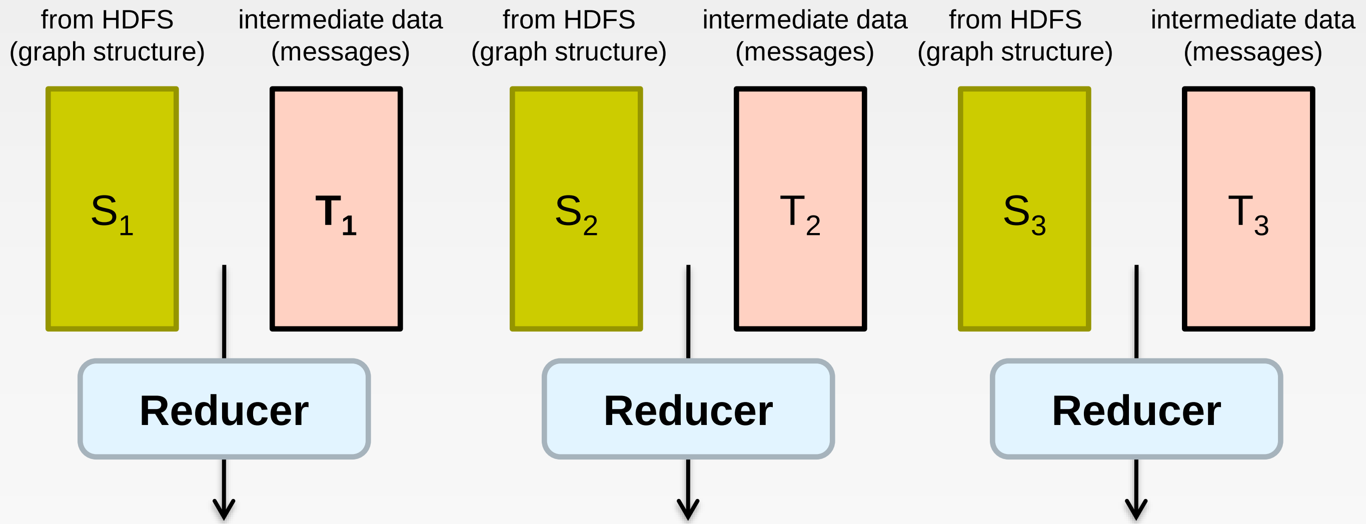


Better Partitioning

- Default: hash partitioning
  - Randomly assign nodes to partitions
- Observation: many graphs exhibit local structure
  - E.g., communities in social networks
  - Better partitioning creates more opportunities for local aggregation
- Unfortunately, partitioning is **hard**!
  - Sometimes, chick-and-egg…
  - But cheap heuristics sometimes available
  - For webgraphs: range partition on domain-sorted URLs

Schimmy Design Pattern

- Basic implementation contains two dataflows:
  - Messages (actual computations)
  - Graph structure (“bookkeeping”)
- Schimmy = reduce side parallel merge join between graph structure and messages
  - Consistent partitioning between input and intermediate data
  - Mappers emit only messages (actual computation)
  - Reducers read graph structure directly from HDFS



References

- Chapter 5. Mining of Massive Datasets.
- Chapter 5. Data-Intensive Text Processing with MapReduce

End of Chapter 9