Map-Reduce Applications: PageRank

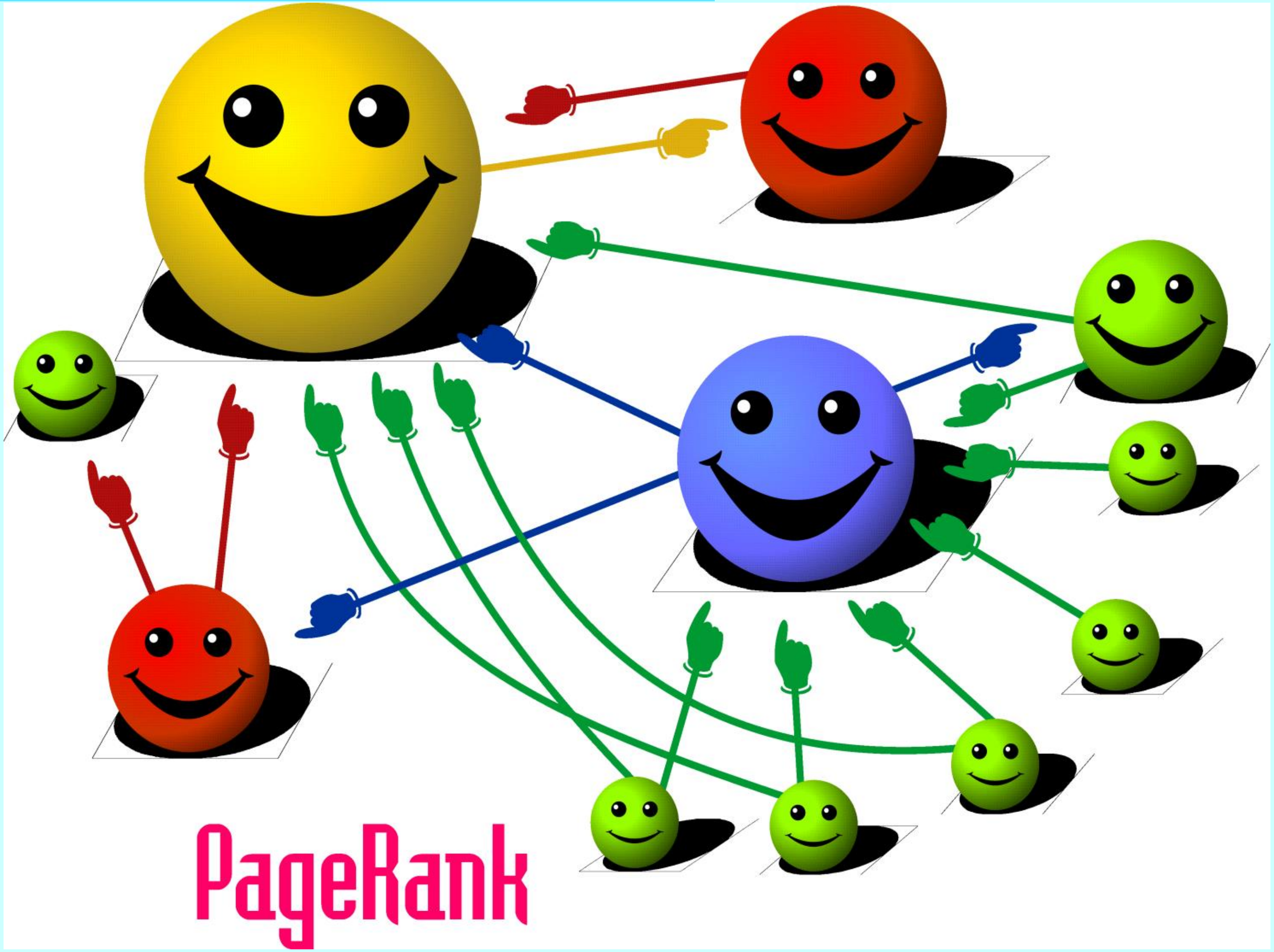
Adapted from Nets212 (Univ. of Pennsylvania) and CS290N courses

Overview

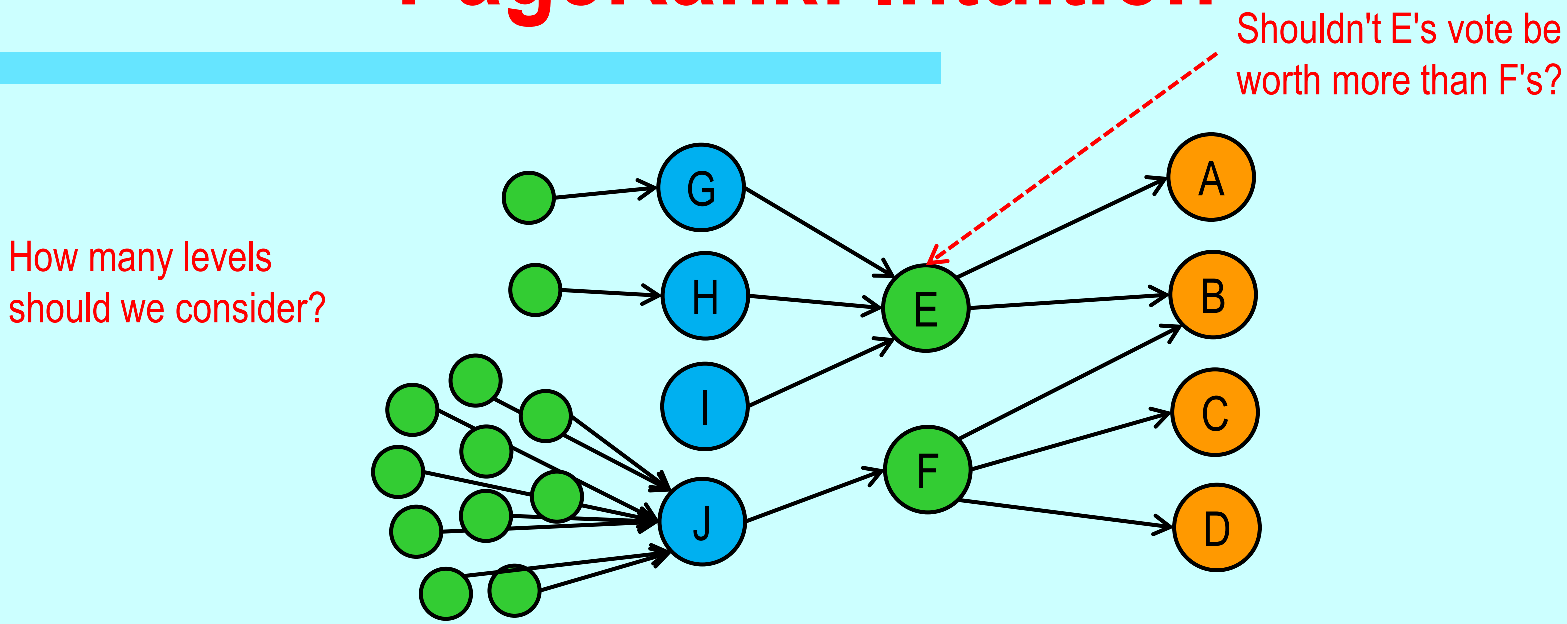
- PageRank
- MapReduce for PageRank

MapReduce: Recap

- Programmers must specify:
  - map (k, v) → <k’, v’>*
  - reduce (k’, v’) → <k’, v’>*
    - All values with the same key are reduced together
- Optionally, also:
  - partition (k’, number of partitions) → partition for k’
    - Often a simple hash of the key, e.g., hash(k’) mod n
    - Divides up key space for parallel reduce operations
  - combine (k’, v’) → <k’, v’>*
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic
- The execution framework handles everything else…


PageRank

PageRank: Intuition



- Imagine a contest for The Web's Best Page
  - Initially, each page has one vote
  - Each page votes for all the pages it has a link to
  - To ensure fairness, pages voting for more than one page must split their vote equally between them
  - Voting proceeds in rounds; in each round, each page has the number of votes it received in the previous round

PageRank

- Each page i is given a rank x_i
- Goal: Assign the x_i such that the rank of each page is governed by the ranks of the pages linking to it:

$$x_i = \sum_{j \in B_i} \frac{1}{N_j} x_j$$

Rank of page i

Every page j that links to i

Number of links out from page j

Rank of page j

How do we compute the rank values?

MapReduce : PageRank

- PageRank models the behavior of a “random surfer”.

$$PR(x) = (1-d) + d\sum_{i=1}^{n}\frac{PR(t_i)}{C(t_i)}$$

- C(t) is the out-degree of t, and (1-d) is a damping factor (random jump)
- The “random surfer” keeps clicking on successive links at random not taking content into consideration.
- Distributes its pages rank equally among all pages it links to.
- The dampening factor takes the surfer “getting bored” and typing arbitrary URL.

Computing PageRank

Start with seed *PageRank* values

Each page distributes *PageRank* "credit" to all pages it points to.

Each target page adds up "credit" from multiple in-bound links to compute PR_{i+1}

PageRank : Key Insights

- Effects at each iteration is local.
  - $i+1^{th}$ iteration depends only on i^{th} iteration
- At iteration i, PageRank for individual nodes can be computed independently

PageRank using MapReduce

- Use Sparse matrix representation (M)
- Map each row of M to a list of PageRank "credit" to assign to out link neighbours.
- These prestige scores are *reduced* to a single PageRank value for a page by aggregating over them.

PageRank: Example

- Show PageRank computation
  - Simple example

Random Surfer Model

- PageRank has an intuitive basis in random walks on graphs
- Imagine a random surfer, who starts on a random page and, in each step,
  - with probability d, clicks on a random link on the page
  - with probability 1-d, jumps to a random page (bored?)
- The PageRank of a page can be interpreted as the fraction of steps the surfer spends on the corresponding page
  - Transition matrix can be interpreted as a Markov Chain

Iterative PageRank (simplified)

Initialize all ranks to be equal, e.g.:

$$x_i^{(0)} = \frac{1}{n}$$

Iterate until convergence

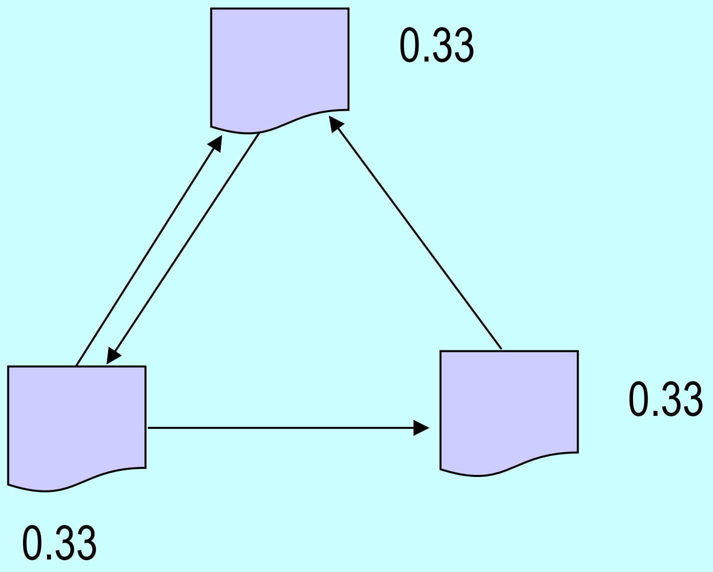
$$x_i^{(k+1)} = \sum_{j \in B_i} \frac{1}{N_j} x_j^{(k)}$$

No need to decide how many levels to consider!

Example: Step 0

Initialize all ranks to be equal

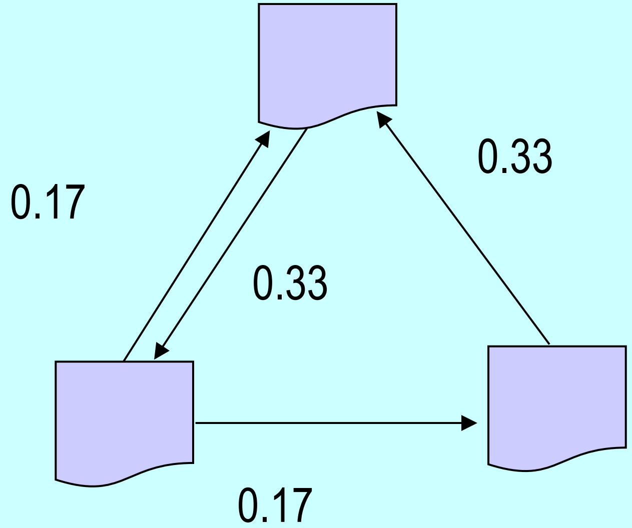
$$x_i^{(0)} = \frac{1}{n}$$



Example: Step 1

Propagate weights across out-edges

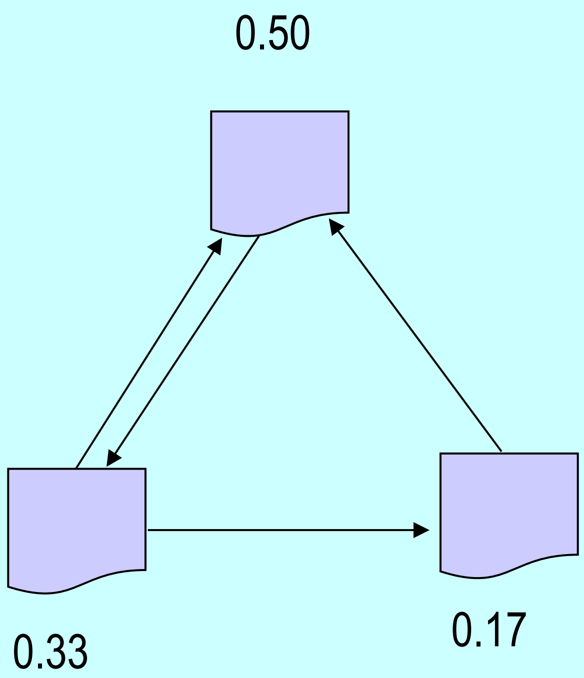
$$x_i^{(k+1)} = \sum_{j \in B_i} \frac{1}{N_j} x_j^{(k)}$$



Example: Step 2

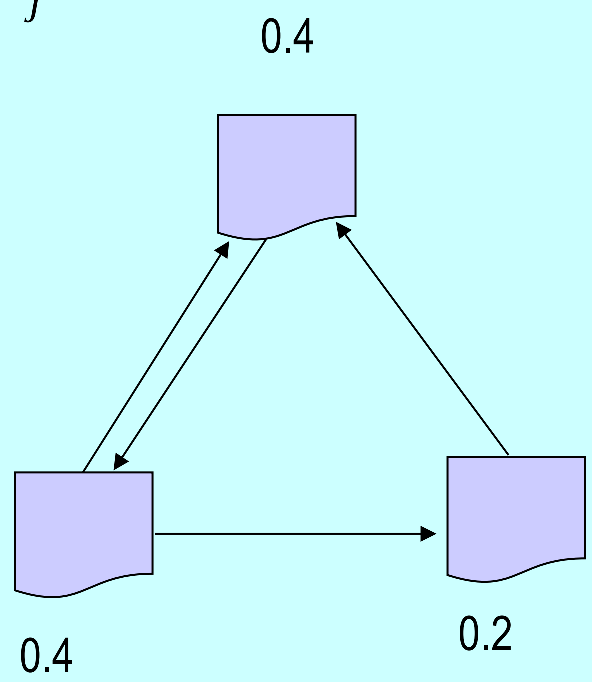
Compute weights
based on in-edges

$$x_i^{(1)} = \sum_{j \in B_i} \frac{1}{N_j} x_j^{(0)}$$



Example: Convergence

$$x_i^{(k+1)} = \sum_{j \in B_i} \frac{1}{N_j} x_j^{(k)}$$



Naïve PageRank Algorithm Restated

- Let
  - N(p) = number outgoing links from page p
  - B(p) = number of back-links to page p

$$PageRank(p) = \sum_{b \in B(p)} \frac{1}{N(b)} PageRank(b)$$

  - Each page b distributes its importance to all of the pages it points to (so we scale by 1/N(b))
  - Page p's importance is increased by the importance of its back set

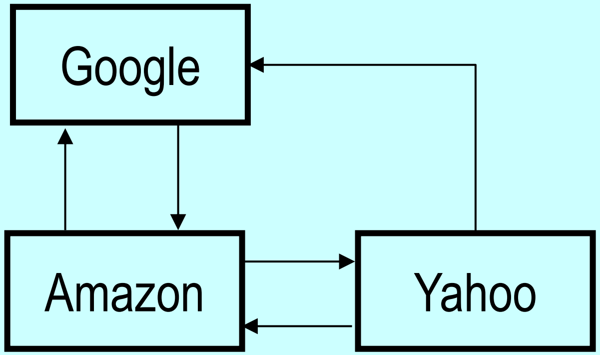
In Linear Algebra formulation

- **Create an m x m matrix M to capture links:**
  - $M(i, j) = 1 / n_j$ if page i is pointed to by page j and page j has n_j outgoing links
    $= 0$ otherwise
  - Initialize all PageRanks to 1, multiply by M repeatedly until all values converge:

$$\begin{bmatrix} PageRank(p_1') \\ PageRank(p_2') \\ \ldots \\ PageRank(p_m') \end{bmatrix} = M \begin{bmatrix} PageRank(p_1) \\ PageRank(p_2) \\ \ldots \\ PageRank(p_m) \end{bmatrix}$$

  - Computes principal eigenvector via power iteration

A brief example



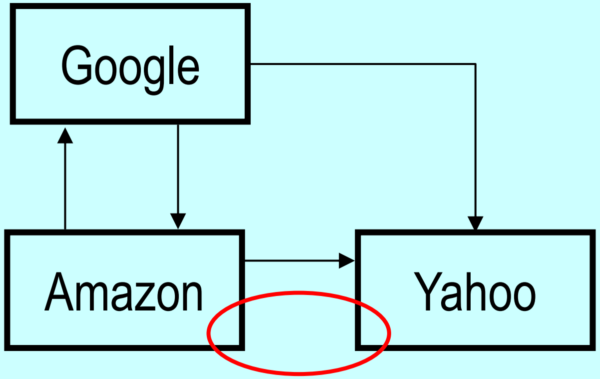
$$\begin{vmatrix} g' \\ y' \\ a' \end{vmatrix} = \begin{vmatrix} 0 & 0.5 & 0.5 \\ 0 & 0 & 0.5 \\ 1 & 0.5 & 0 \end{vmatrix} * \begin{vmatrix} g \\ y \\ a \end{vmatrix}$$

Running for multiple iterations:

$$\begin{vmatrix} g \\ y \\ a \end{vmatrix} = \begin{vmatrix} 1 \\ 1 \\ 1 \end{vmatrix}, \begin{vmatrix} 1 \\ 0.5 \\ 1.5 \end{vmatrix}, \begin{vmatrix} 1 \\ 0.75 \\ 1.25 \end{vmatrix}, \ldots \begin{vmatrix} 1 \\ 0.67 \\ 1.33 \end{vmatrix}$$

Total rank sums to number of pages

Oops #1 – PageRank sinks



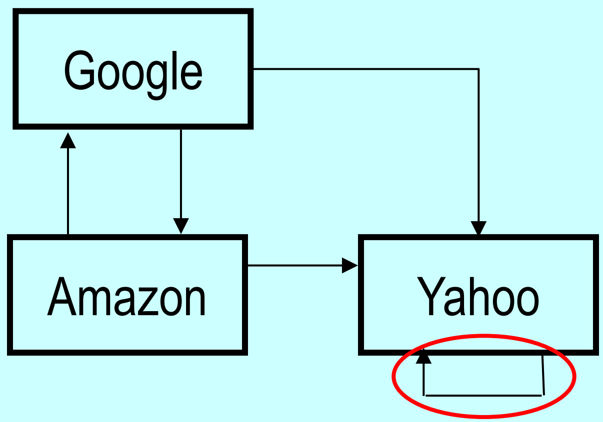
$$\begin{vmatrix} g' \\ y' \\ a' \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0.5 \\ 0.5 & 0 & 0.5 \\ 0.5 & 0 & 0 \end{vmatrix} * \begin{vmatrix} g \\ y \\ a \end{vmatrix}$$

'dead end' - PageRank is lost after each round

Running for multiple iterations:

$$\begin{vmatrix} g \\ y \\ a \end{vmatrix} = \begin{vmatrix} 1 \\ 1 \\ 1 \end{vmatrix}, \begin{vmatrix} 0.5 \\ 1 \\ 0.5 \end{vmatrix}, \begin{vmatrix} 0.25 \\ 0.5 \\ 0.25 \end{vmatrix}, \ldots, \begin{vmatrix} 0 \\ 0 \\ 0 \end{vmatrix}$$

Oops #2 – PageRank hogs



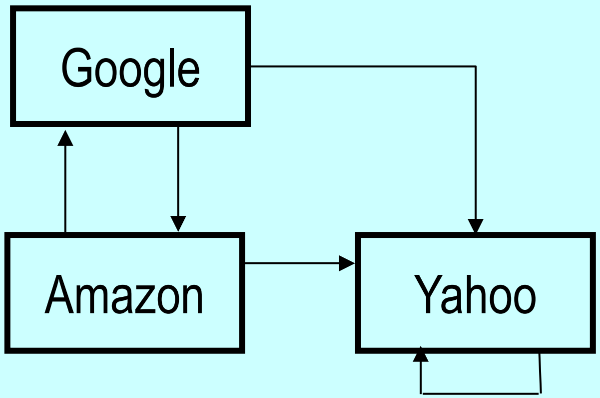
PageRank cannot flow out and accumulates

$$\begin{vmatrix} g' \\ y' \\ a' \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0.5 \\ 0.5 & 1 & 0.5 \\ 0.5 & 0 & 0 \end{vmatrix} * \begin{vmatrix} g \\ y \\ a \end{vmatrix}$$

Running for multiple iterations:

$$\begin{vmatrix} g \\ y \\ a \end{vmatrix} = \begin{vmatrix} 1 \\ 1 \\ 1 \end{vmatrix}, \begin{vmatrix} 0.5 \\ 2 \\ 0.5 \end{vmatrix}, \begin{vmatrix} 0.25 \\ 2.5 \\ 0.25 \end{vmatrix}, \ldots, \begin{vmatrix} 0 \\ 3 \\ 0 \end{vmatrix}$$

Stopping the Hog



$$\begin{vmatrix} g' \\ y' \\ a' \end{vmatrix} = 0.85 \begin{vmatrix} 0 & 0 & 0.5 \\ 0.5 & 1 & 0.5 \\ 0.5 & 0 & 0 \end{vmatrix} * \begin{vmatrix} g \\ y \\ a \end{vmatrix} + \begin{vmatrix} 0.15 \\ 0.15 \\ 0.15 \end{vmatrix}$$

Running for multiple iterations:

$$\begin{vmatrix} g \\ y \\ a \end{vmatrix} = \begin{vmatrix} 0.57 \\ 1.85 \\ 0.57 \end{vmatrix}, \begin{vmatrix} 0.39 \\ 2.21 \\ 0.39 \end{vmatrix}, \begin{vmatrix} 0.32 \\ 2.36 \\ 0.32 \end{vmatrix}, \ldots, \begin{vmatrix} 0.26 \\ 2.48 \\ 0.26 \end{vmatrix}$$

… though does this seem right?

Improved PageRank

- Remove out-degree 0 nodes (or consider them to refer back to referrer)
- Add decay factor d to deal with sinks

$$PageRank(p) = (1-d) + d \sum_{b \in B_p} \frac{1}{N(b)} PageRank(b)$$

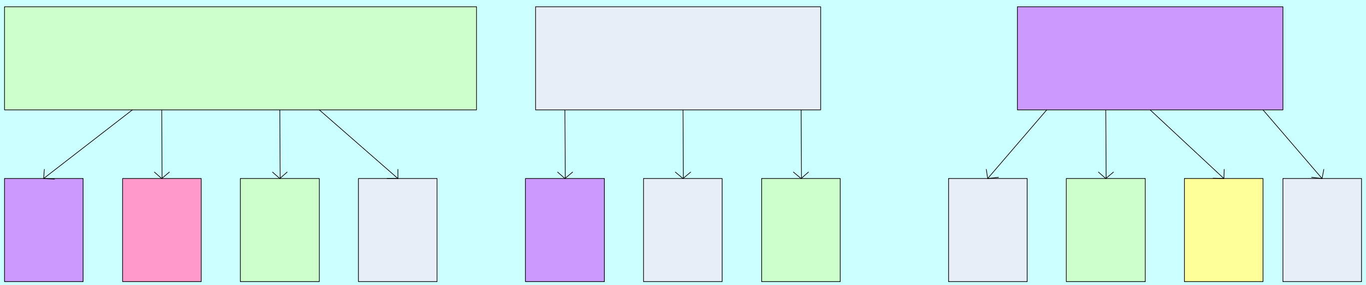
- Typical value: d=0.85
- Intuition in the idea of the "random surfer":
  - Surfer occasionally stops following link sequence and jumps to new random page, with probability 1 - *d*

PageRank on MapReduce

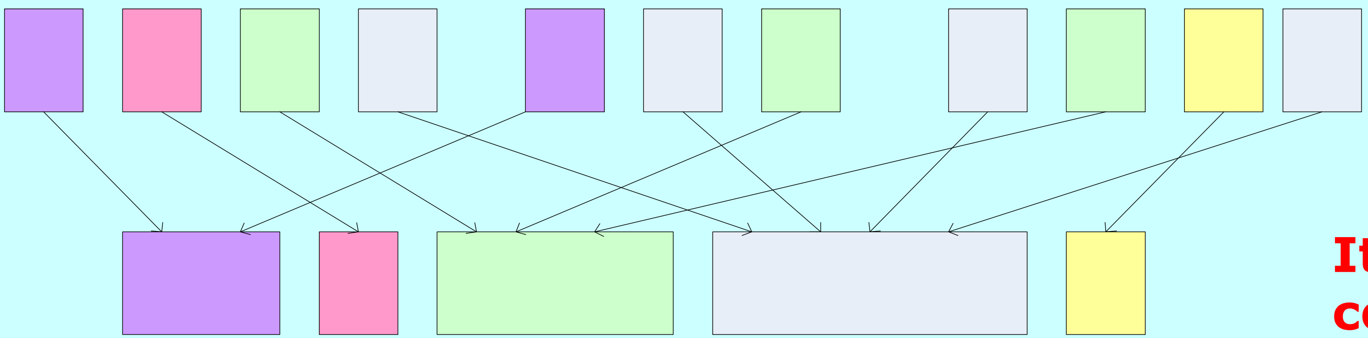
- **Inputs**
  - Of the form: page → (currentWeightOfPage, {adjacency list})
- **Map**
  - Page p "propagates" $1/N_p$ of its d * weight(p) to the destinations of its out-edges (think like a vertex!)
- **Reduce**
  - p-th page sums the incoming weights and adds (1-d), to get its weight'(p)
- **Iterate until convergence**
  - Common practice: run some fixed number of times, e.g., 25x
  - Alternatively: Test after each iteration with a second MapReduce job, to determine the maximum change between old and new weights

PageRank using MapReduce

Map: distribute PageRank "credit" to link targets



Reduce: gather up PageRank "credit" from multiple sources to compute new PageRank value



Iterate until convergence

Source of Image: Lin 2008

Phase 1: Process HTML

- Map task takes (URL, page-content) pairs and maps them to (URL, (PR_{init}, list-of-urls))
  - PR_{init} is the "seed" PageRank for URL
  - list-of-urls contains all pages pointed to by URL
- Reduce task is just the identity function

Phase 2: PageRank Distribution

- Reduce task gets (URL, url_list) and many (URL, *val*) values
  - Sum *val*s and fix up with *d to get new PR*
  - Emit (URL, (new_rank, url_list))

- Check for convergence using non parallel component

PageRank Calculation: Preliminaries

One PageRank iteration:

- Input:
  - $(id_1, [score_1^{(t)}, out_{11}, out_{12}, ..]), (id_2, [score_2^{(t)}, out_{21}, out_{22}, ..])$ ..
- Output:
  - $(id_1, [score_1^{(t+1)}, out_{11}, out_{12}, ..]), (id_2, [score_2^{(t+1)}, out_{21}, out_{22}, ..])$ ..

MapReduce elements

- Score distribution and accumulation
- Database join
- Side-effect files

PageRank:
Score Distribution and Accumulation

- Map
  - In: $(id_1, [score_1^{(t)}, out_{11}, out_{12}, ..])$, $(id_2, [score_2^{(t)}, out_{21}, out_{22}, ..])$ ..
  - Out: $(out_{11}, score_1^{(t)}/n_1)$, $(out_{12}, score_1^{(t)}/n_1)$ .., $(out_{21}, score_2^{(t)}/n_2)$, ..
- Shuffle & Sort by node_id
  - In: $(id_2, score_1)$, $(id_1, score_2)$, $(id_1, score_1)$, ..
  - Out: $(id_1, score_1)$, $(id_1, score_2)$, .., $(id_2, score_1)$, ..
- Reduce
  - In: $(id_1, [score_1, score_2, ..])$, $(id_2, [score_1, ..])$, ..
  - Out: $(id_1, score_1^{(t+1)})$, $(id_2, score_2^{(t+1)})$, ..

PageRank:
Database Join to associate outlinks with score

- Map
  - In & Out: (id_1, $score_1^{(t+1)}$), (id_2, $score_2^{(t+1)}$), .., (id_1, [out_{11}, out_{12}, ..]), (id_2, [out_{21}, out_{22}, ..]) ..
- Shuffle & Sort by node_id
  - Out: (id_1, $score_1^{(t+1)}$), (id_1, [out_{11}, out_{12}, ..]), (id_2, [out_{21}, out_{22}, ..]), (id_2, $score_2^{(t+1)}$), ..
- Reduce
  - In: (id_1, [$score_1^{(t+1)}$, out_{11}, out_{12}, ..]), (id_2, [out_{21}, out_{22}, .., $score_2^{(t+1)}$]), ..
  - Out: (id_1, [$score_1^{(t+1)}$, out_{11}, out_{12}, ..]), (id_2, [$score_2^{(t+1)}$, out_{21}, out_{22}, ..]) ..

PageRank: Side Effect Files for dangling nodes

- Dangling Nodes
  - Nodes with no outlinks (observed but not crawled URLs)
  - Score has no outlet
    - need to distribute to all graph nodes evenly
- Map for dangling nodes:
  - In: .., (id_3, [$score_3$]), ..
  - Out: .., ("*", 0.85×$score_3$), ..
- Reduce
  - In: .., ("*", [$score_1$, $score_2$, ..]), ..
  - Out: .., everything else, ..
  - Output to side-effect: ("*", score), fed to Mapper of next iteration