MapReduce and Graph Data

Why Graphs

- Graphs can be used to represent many Data problems today:
  - Social networks: Facebook, Twitter, etc
  - Communication networks
  - Biological networks
  - Transportation, road networks
  - …
- Many of these problems deal with Big Data
  - So, use MapReduce (or in general cluster computing)

Graph DBs

- Graph databases getting attention!
  - http://www.forbes.com/sites/danwoods/2014/02/14/50-shades-of-graph-how-graph-databases-are-transforming-online-dating/
  - http://www.pcworld.com/article/2361140/graph-databases-find-answers-for-the-sick-and-their-healers.html

Graph Algorithms in MapReduce

- G = (V,E), where
  - V represents the set of vertices (nodes)
  - E represents the set of edges (links)
  - Both vertices and edges may contain additional information

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
- Key questions:
  - How do you represent graph data in MapReduce?
  - How do you traverse a graph in MapReduce?

Representing Graphs

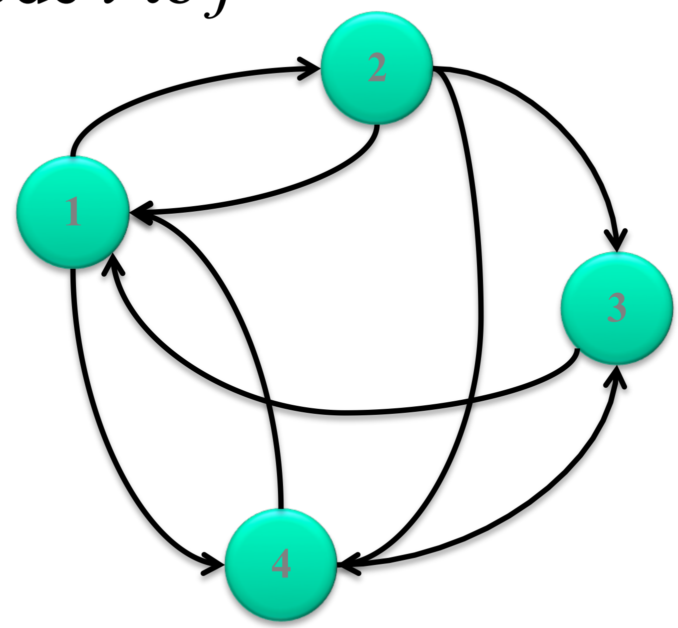
- $G = (V, E)$
- Two common representations
  - Adjacency matrix
  - Adjacency list

Adjacency Matrices

Represent a graph as an n x n square matrix M

- $n = |V|$
- $M_{ij} = 1$ means a link from node i to j

	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0



Adjacency Matrices: Critique

- Advantages:
  - Amenable to mathematical manipulation
  - Iteration over rows and columns corresponds to computations on outlinks and inlinks
- Disadvantages:
  - Lots of zeros for sparse matrices
  - Lots of wasted space

Adjacency Lists

Take adjacency matrices… and throw away all the zeros

	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0

1: 2, 4
2: 1, 3, 4
3: 1
4: 1, 3

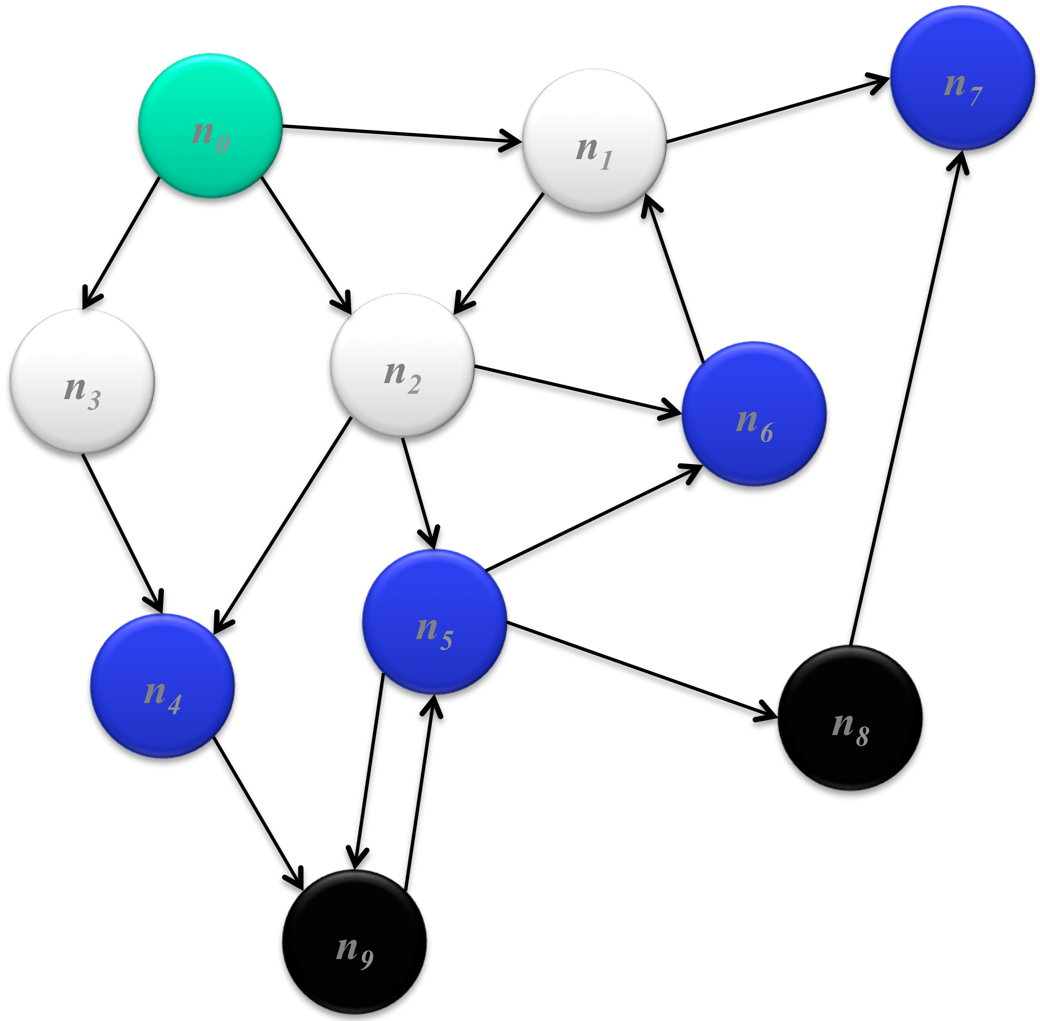
Adjacency Lists: Critique

- Advantages:
  - Much more compact representation
  - Easy to compute over outlinks
- Disadvantages:
  - Much more difficult to compute over inlinks

Finding the Shortest Path

- Consider simple case of equal edge weights
- Solution to the problem can be defined inductively
- Here's the intuition:
  - Define: b is reachable from a if b is on adjacency list of a
  - DISTANCETO(s) = 0
  - For all nodes p reachable from s, DISTANCETO(p) = 1
  - For all nodes n reachable from some other set of nodes M, DISTANCETO(n) = 1 + min(DISTANCETO(m), $m \in M$)

Visualizing Parallel BFS



From Intuition to Algorithm

- Data representation:
  - Key: node n
  - Value: d (distance from start), adjacency list (list of nodes reachable from n)
  - Initialization: for all nodes except for start node, $d = \infty$
- Mapper:
  - $\forall m \in$ adjacency list: emit (m, d + 1)
- Sort/Shuffle
  - Groups distances by reachable nodes
- Reducer:
  - Selects minimum distance path for each reachable node
  - Additional bookkeeping needed to keep track of actual path

Multiple Iterations Needed

- Each MapReduce iteration advances the "known frontier" by one hop
  - Subsequent iterations include more and more reachable nodes as frontier expands
  - Multiple iterations are needed to explore entire graph
- Preserving graph structure:
  - Problem: Where did the adjacency list go?
  - Solution: mapper emits (*n*, adjacency list) as well

BFS Pseudo-Code

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

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

Stopping Criterion

- How many iterations are needed in parallel BFS (equal edge weight case)?
- Convince yourself: when a node is first "discovered", we've found the shortest path
- Now answer the question...
  - Six degrees of separation?

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

Random Walks Over the Web

- Random surfer model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- PageRank
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages
- PageRank captures notions of page importance
  - One of thousands of features used in web search
  - Note: query-independent

PageRank: Defined

Given page x with inlinks $t_1 \ldots t_n$, where

- $C(t)$ is the out-degree of t
- α is probability of random jump
- N is the total number of nodes in the graph

$$PR(x) = \alpha\left(\frac{1}{N}\right) + (1-\alpha)\sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}$$

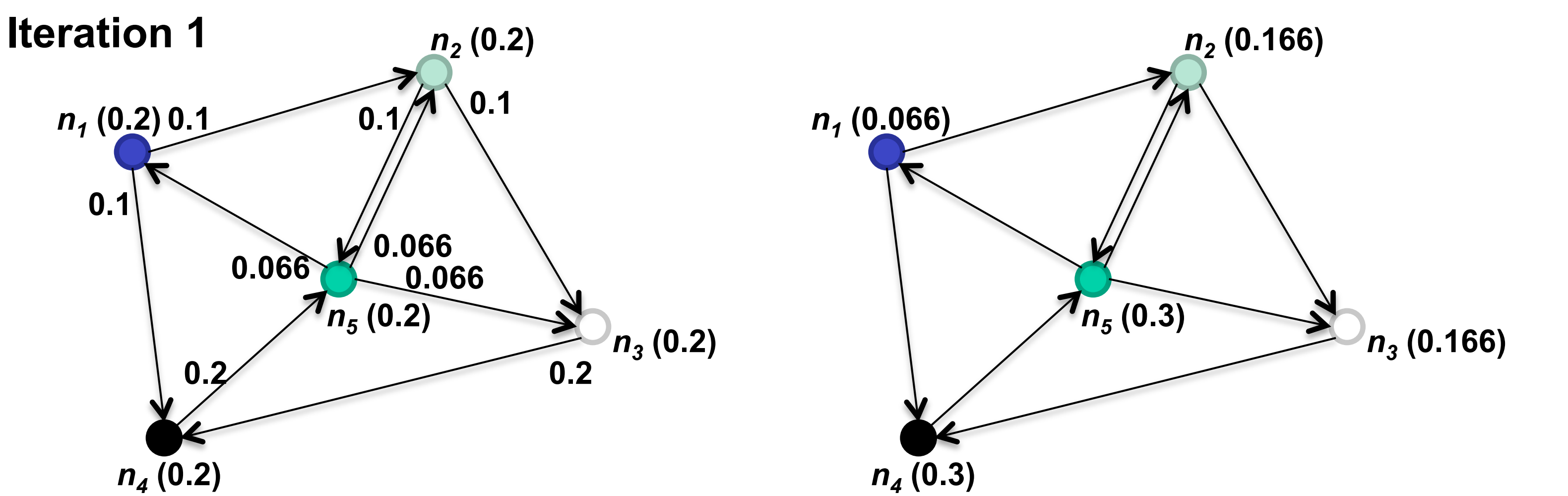
Computing PageRank

- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm:
  - Start with seed PR_i values
  - Each page distributes PR_i "credit" to all pages it links to
  - Each target page adds up "credit" from multiple in-bound links to compute PR_{i+1}
  - Iterate until values converge

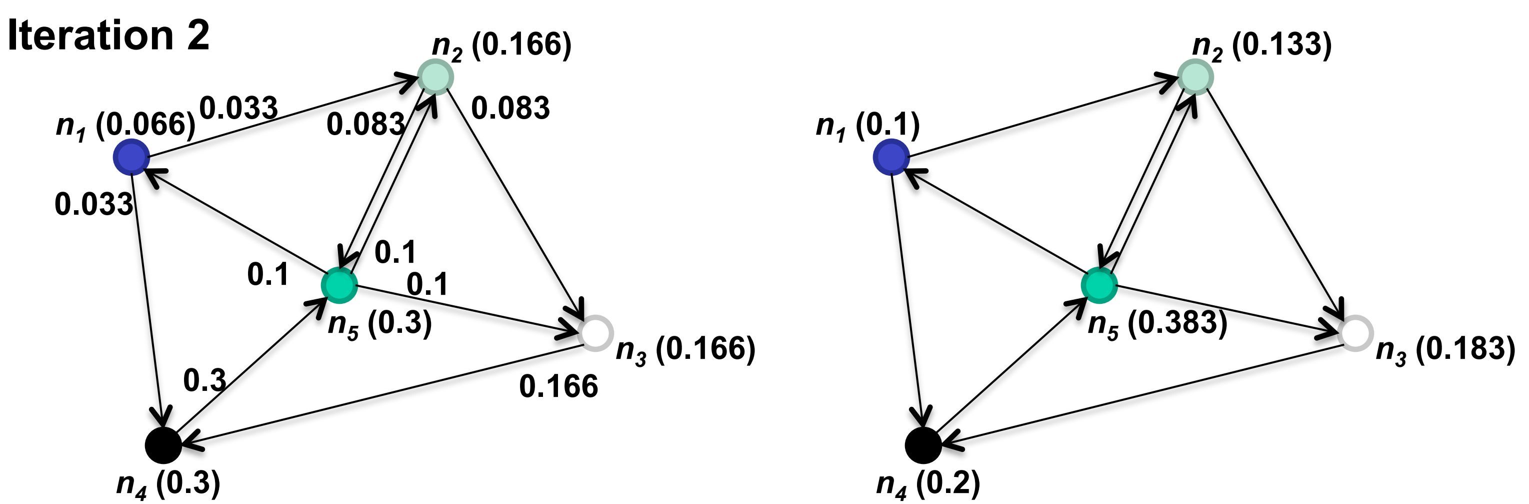
Simplified PageRank

- First, tackle the simple case:
  - No random jump factor
  - No dangling links
- Then, factor in these complexities…
  - Why do we need the random jump?
  - Where do dangling links come from?

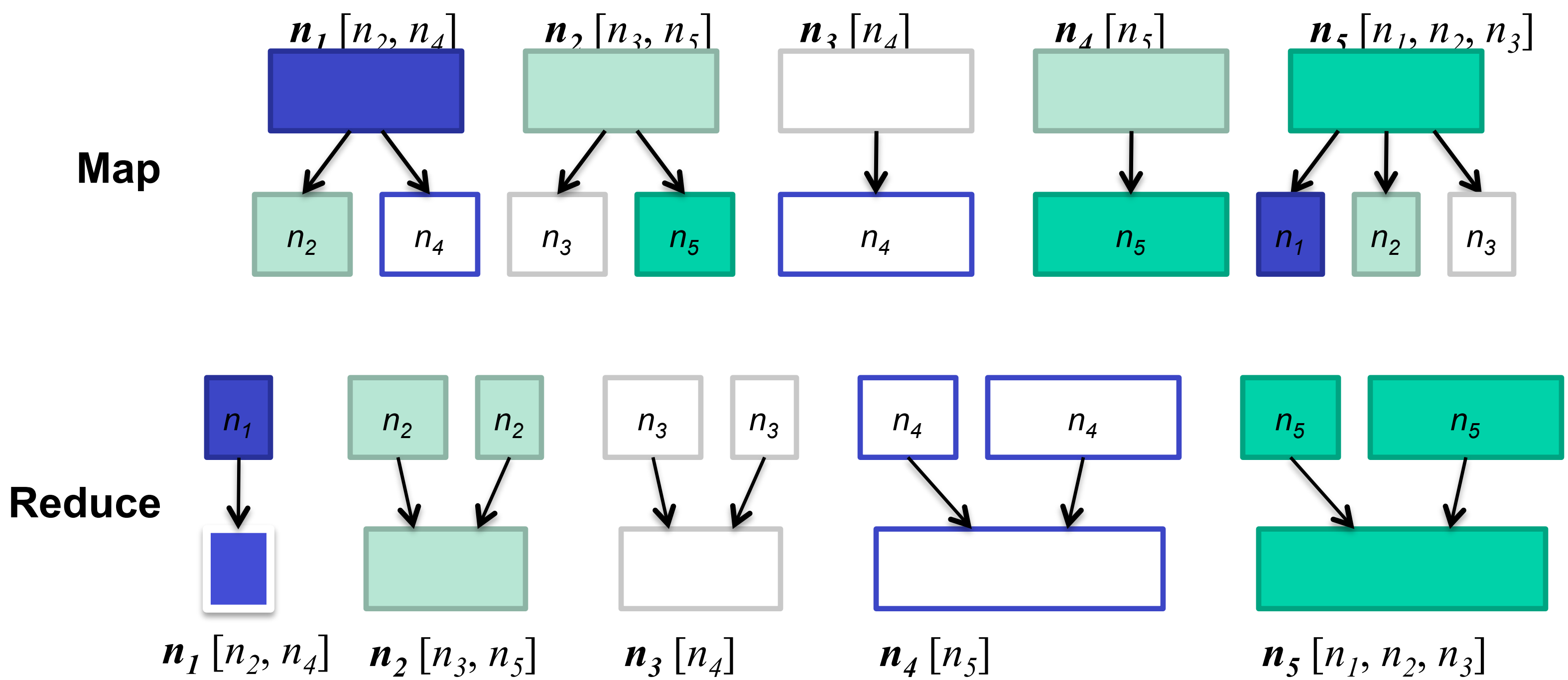
Sample PageRank Iteration (1)



Sample PageRank Iteration (2)



PageRank in MapReduce



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:
  - Second pass to redistribute “missing PageRank mass” and account for random jumps

$$p' = \alpha\left(\frac{1}{|G|}\right) + (1-\alpha)\left(\frac{m}{|G|} + p\right)$$

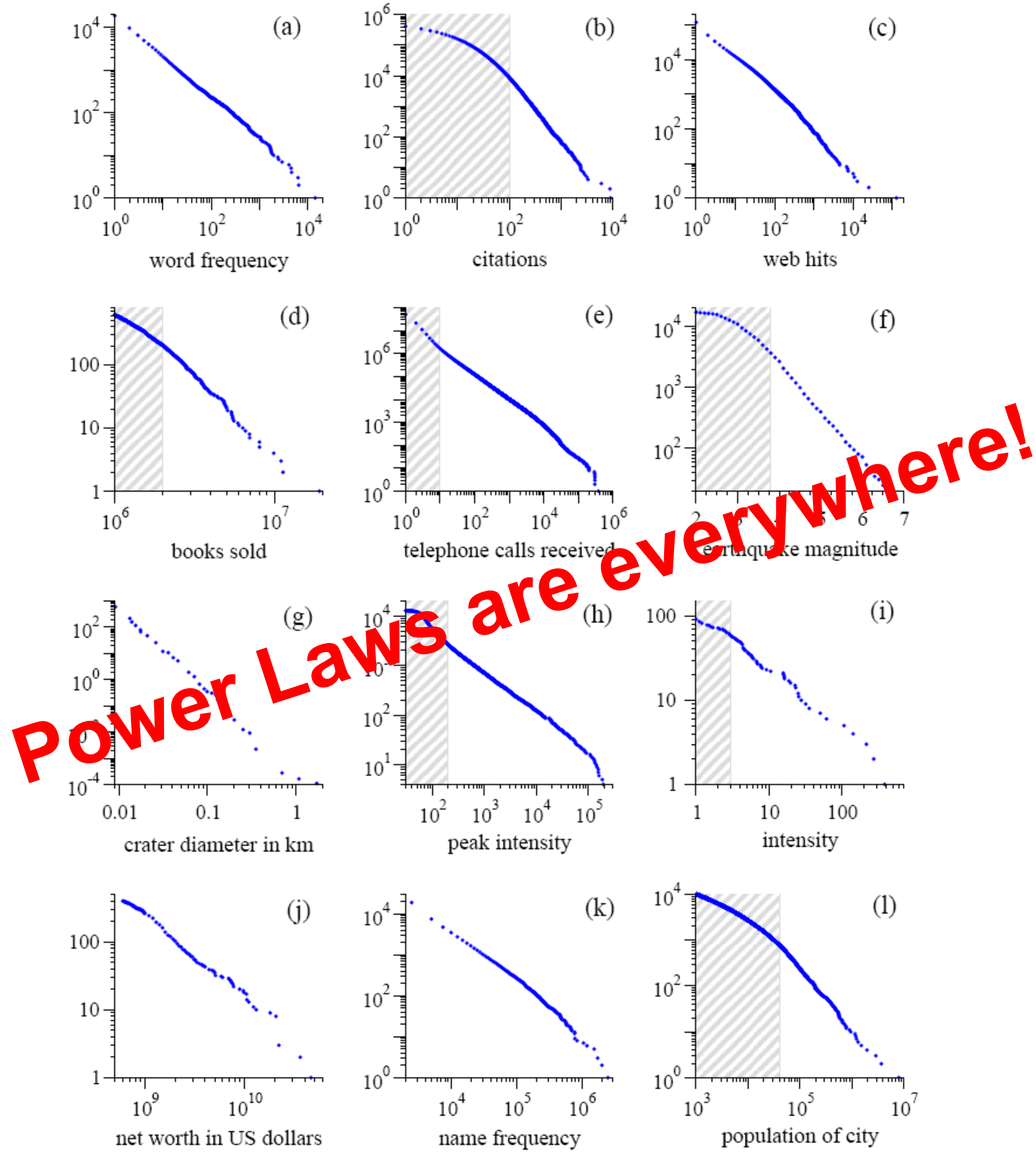
  - p is PageRank value from before, p' is updated PageRank value
  - |G| is the number of nodes in the graph
  - m is the missing PageRank mass

PageRank Convergence

- Alternative convergence criteria
  - Iterate until PageRank values don’t change
  - Iterate until PageRank rankings don’t change
  - Fixed number of iterations
- Convergence for web graphs?

Efficient Graph Algorithms

- Sparse vs. dense graphs
- Graph topologies



Power Laws are everywhere!
(a)
word frequency
(b)
citations
(c)
web hits
(d)
books sold
(e)
telephone calls received
(f)
earthquake magnitude
(g)
crater diameter in km
(h)
peak intensity
(i)
intensity
(j)
net worth in US dollars
(k)
name frequency
(l)
population of city

Local Aggregation

- Use combiners!
  - In-mapper combining design pattern also applicable
- Maximize opportunities for local aggregation
  - Simple tricks: sorting the dataset in specific ways