COMP9313: Big Data Management

Lecturer: Xin Cao

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

Chapter 8.1: Graph Data Management

Graph Data Processing in MapReduce

What's a Graph?

- 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
- Different types of graphs:
  - Directed vs. undirected edges
  - Presence or absence of cycles
- Graphs are everywhere:
  - Hyperlink structure of the Web
  - Physical structure of computers on the Internet
  - Interstate highway system
  - Social networks

Graph Analytics

- General Graph
  - Count the number of nodes whose degree is equal to 5
  - Find the diameter of the graphs
- Web Graph
  - Rank each webpage in the web graph or each user in the twitter graph using PageRank, or other centrality measure
- Transportation Network
  - Return the shortest or cheapest flight/road from one city to another
- Social Network
  - Detect a group of users who have similar interests
- Financial Network
  - Find the path connecting two suspicious transactions;
- … …

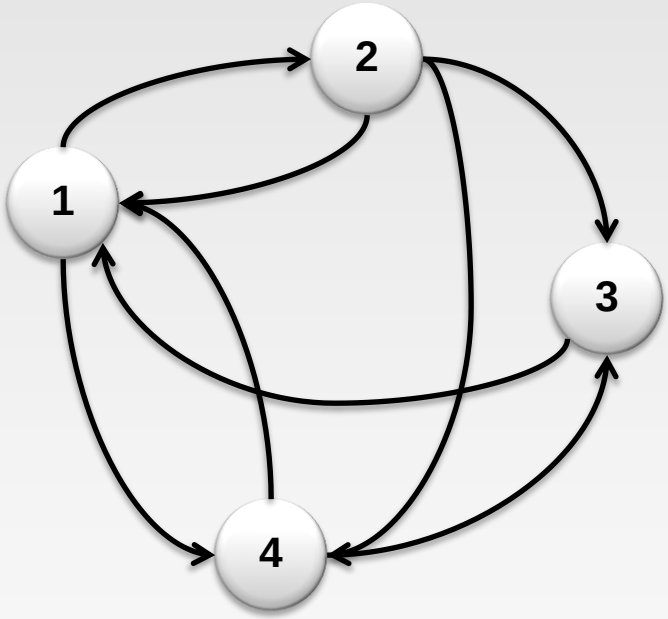
Graphs and MapReduce

- 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

- 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

Representing Graphs

- 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

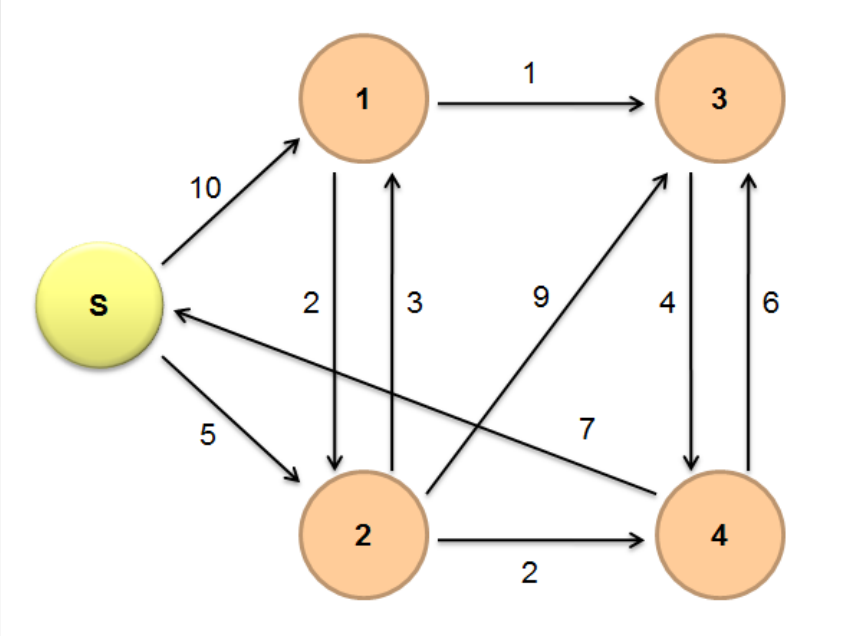
Adjacency Lists: Critique

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

Single-Source Shortest Path

Single-Source Shortest Path (SSSP)

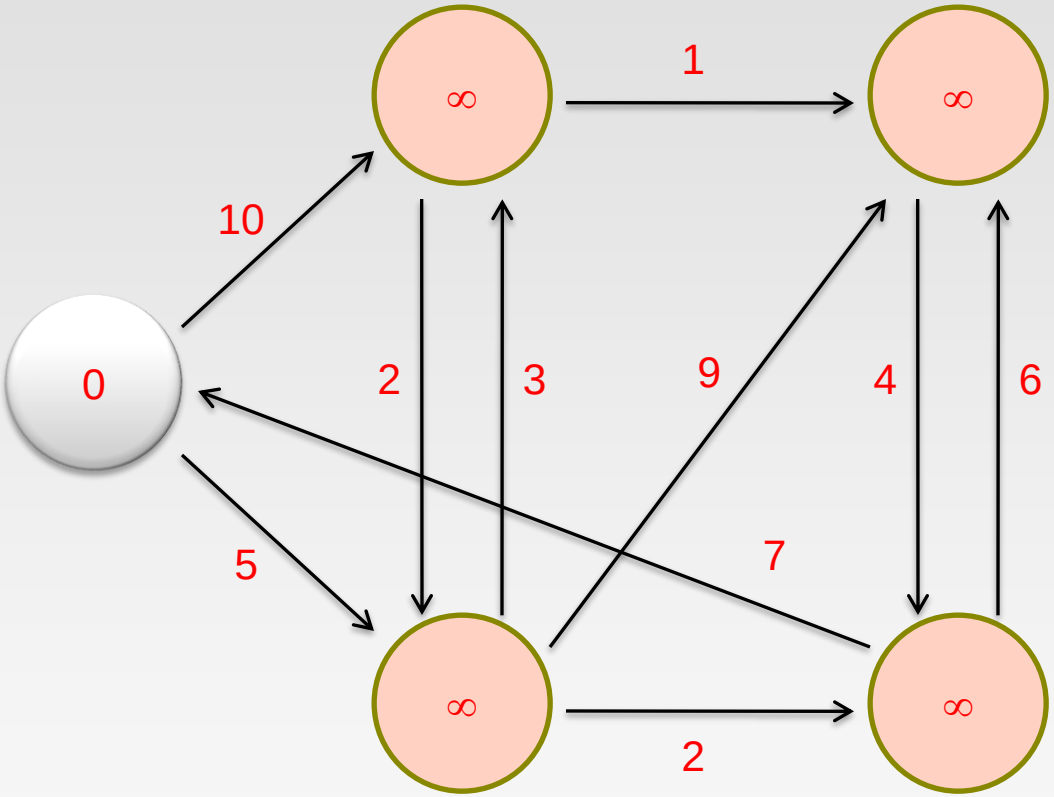
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Dijkstra's Algorithm:
  - For a given source node in the graph, the algorithm finds the shortest path between that node and every other



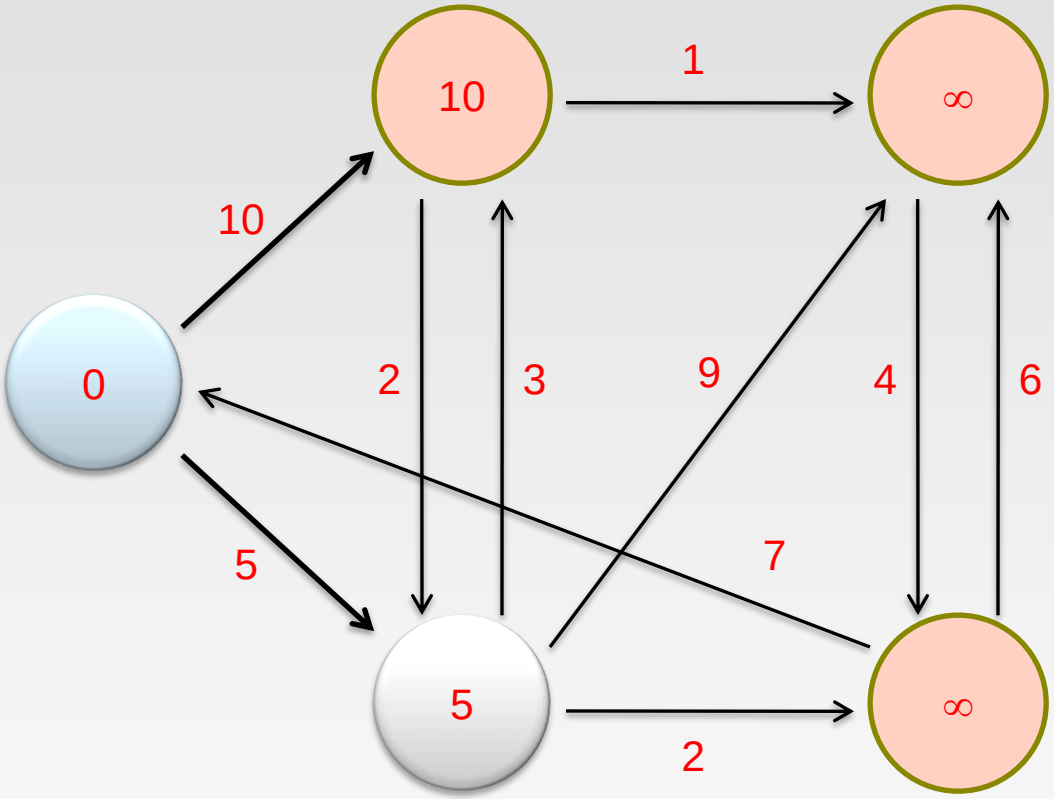
Dijkstra's Algorithm

```
1:  DIJKSTRA(G, w, s)
2:      d[s] ← 0
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)
```

Dijkstra's Algorithm Example

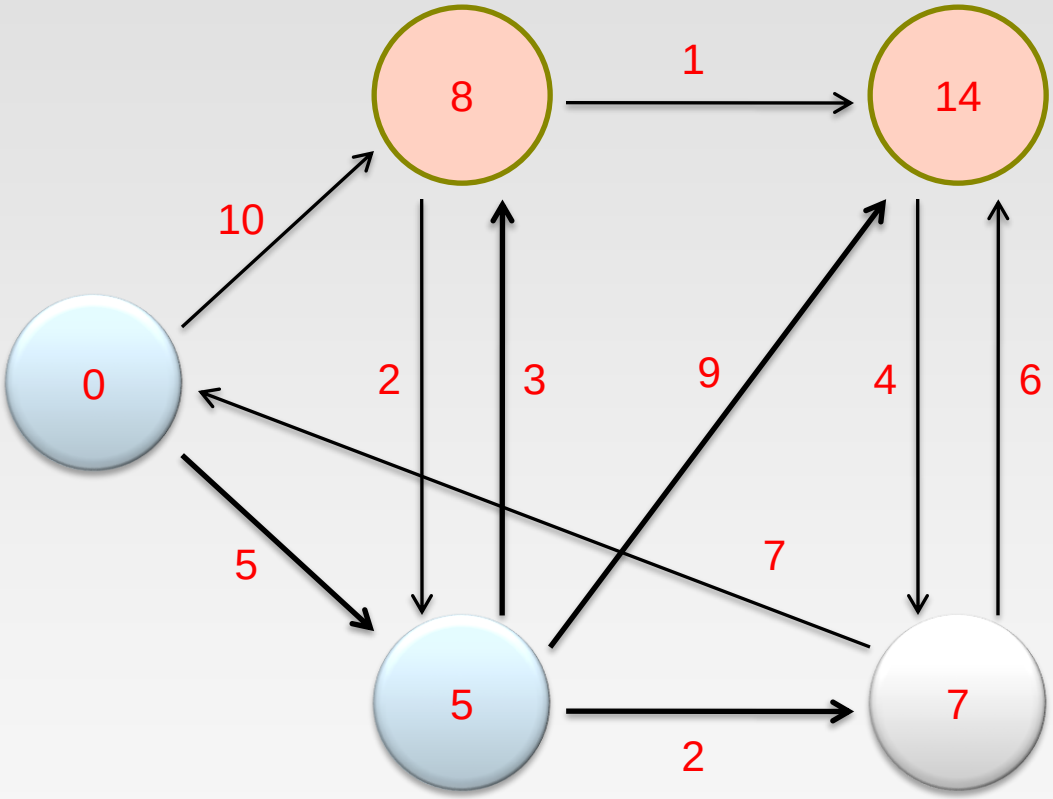


Dijkstra's Algorithm Example

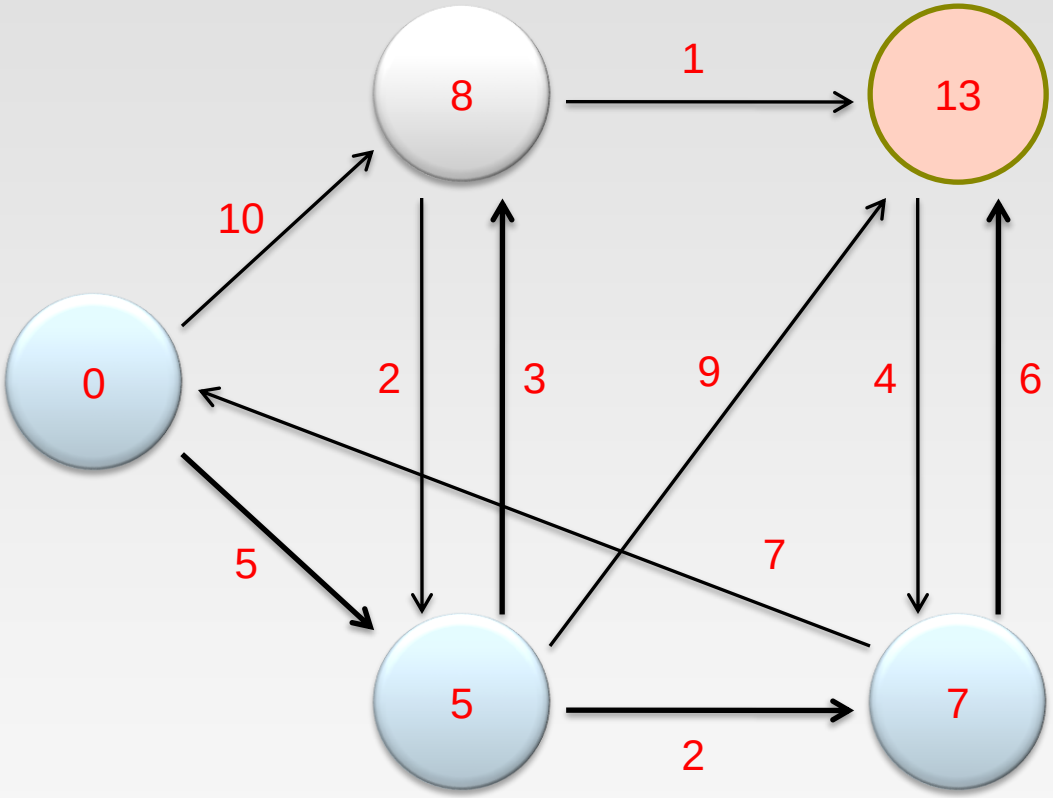


Example from CLR

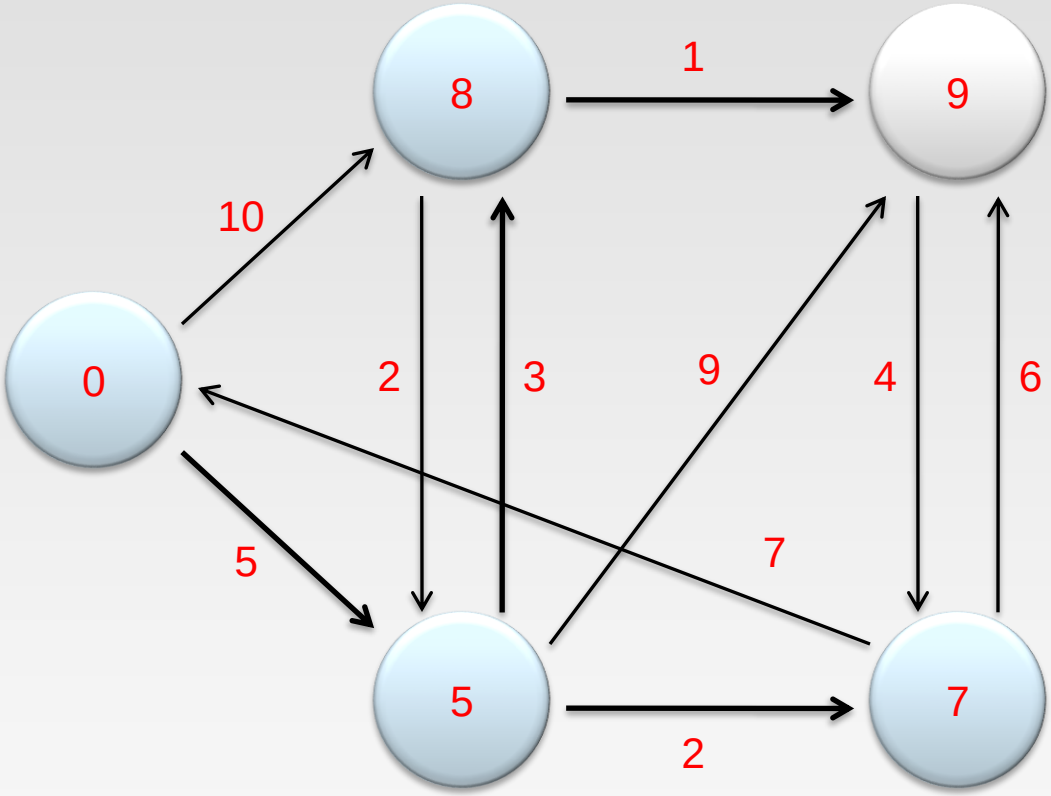
Dijkstra's Algorithm Example



Dijkstra's Algorithm Example

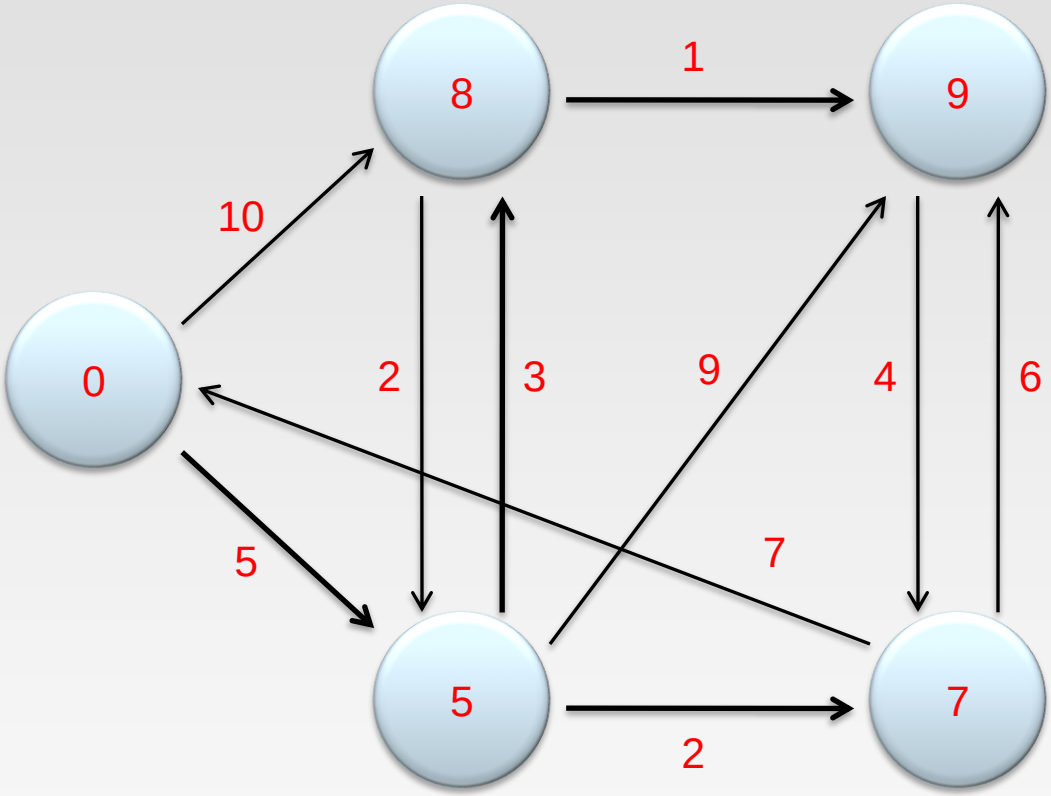


Dijkstra's Algorithm Example



Example from CLR

Dijkstra's Algorithm Example



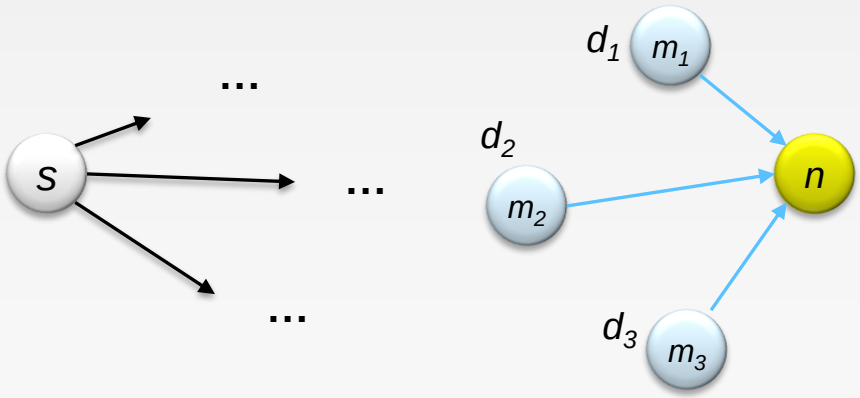
Finish!

Single Source Shortest Path

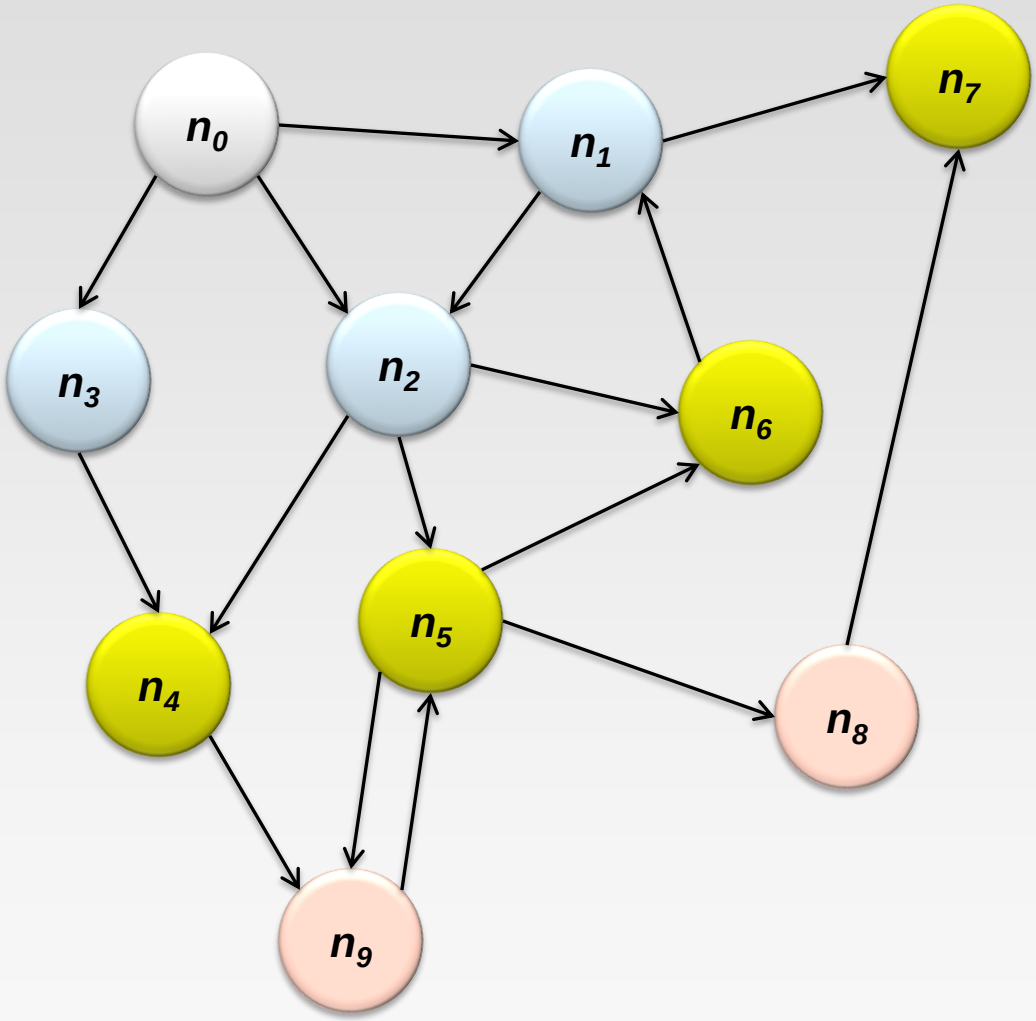
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Single processor machine: Dijkstra's Algorithm
- MapReduce: parallel Breadth-First Search (BFS)

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
  - The input of Mapper is the output of Reducer in the previous iteration
  - 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

- Equal Edge Weights (how to deal with weighted edges?)
- Only distances, no paths stored (how to obtain paths?)

```
class Mapper
    method Map(nid n, node N)
    d ← N.Distance
    Emit(nid n,N.AdjacencyList)                    //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, [d1, d2, . . .])
    d_min←∞
    M ← ∅
    for all d ∈ counts [d1, d2, . . .] do
        if IsNode(d) then
            M.AdjacencyList ← 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...
  - The diameter of the graph, or the greatest distance between any pair of nodes
  - Six degrees of separation?
    - If this is indeed true, then parallel breadth-first search on the global social network would take at most six MapReduce iterations.

Implementation in MapReduce

- The actual checking of the termination condition must occur outside of MapReduce.
- The driver (main) checks to see if a termination condition has been met, and if not, repeats.
- Hadoop provides a lightweight API called “counters”.
  - It can be used for counting events that occur during execution, e.g., number of corrupt records, number of times a certain condition is met, or anything that the programmer desires.
  - Counters can be designed to count the number of nodes that have distances of ∞ at the end of the job, the driver program can access the final counter value and check to see if another iteration is necessary.

Chained MapReduce Job (Java)

- In the main function, you can configure like:

```
String input = IN;
String output = OUT + System.nanoTime();
boolean isdone = false;
while (isdone == false) {
        Job job = Job.getInstance(conf, "traverse job");
        //configure your jobs here such as mapper and reducer classes

        FileInputFormat.addInputPath(job, new Path(input));
        FileOutputFormat.setOutputPath(job, new Path(output));

        job.waitForCompletion(true);        //start the job

        Counters counters = job.getCounters();
        Counter counter = counters.findCounter(MY_COUNTERS.REACHED);

        if(counter.getValue() == 0){        //use the counter to check the termination
                isdone = true;
        }
        input = output;                     //make the current output as the next input
        output = OUT + System.nanoTime();
}
```

https://github.com/himank/Graph-Algorithm-MapReduce/blob/master/src/DijikstraAlgo.java

MapReduce Counters

- Instrument Job's metrics
  - Gather statistics
    - Quality control – confirm what was expected.
      - E.g., count invalid records
    - Application-level statistics.
  - Problem diagnostics
  - Try to use counters for gathering statistics instead of log files
- Framework provides a set of built-in metrics
  - For example, bytes processed for input and output
- User can create new counters
  - Number of records consumed
  - Number of errors or warnings

Built-in Counters

- Hadoop maintains some built-in counters for every job.
- Several groups for built-in counters
  - File System Counters – number of bytes read and written
  - Job Counters – documents number of map and reduce tasks launched, number of failed tasks
  - Map-Reduce Task Counters– mapper, reducer, combiner input and output records counts, time and memory statistics

User-Defined Counters

- You can create your own counters
  - Counters are defined by a Java enum
    - serves to group related counters
    - E.g.,

```
enum Temperature {
        MISSING,
        MALFORMED
}
```

- Increment counters in Reducer and/or Mapper classes
  - Counters are global: Framework accurately sums up counts across all maps and reduces to produce a grand total at the end of the job

Implement User-Defined Counters

- Retrieve Counter from Context object
  - Framework injects Context object into map and reduce methods
- Increment Counter's value
  - Can increment by 1 or more

```
parser.parse(value);
if (parser.isValidTemperature()) {
  int airTemperature = parser.getAirTemperature();
  context.write(new Text(parser.getYear()),
      new IntWritable(airTemperature));
} else if (parser.isMalformedTemperature()) {
  System.err.println("Ignoring possibly corrupt input: " + value);
  context.getCounter(Temperature.MALFORMED).increment(1);
} else if (parser.isMissingTemperature()) {
  context.getCounter(Temperature.MISSING).increment(1);
}
```

Implement User-Defined Counters

- Get Counters from a finished job in Java
  - Counter counters = job.getCounters()
- Get the counter according to name
  - Counter c1 = counters.findCounter(Temperature.MISSING)
- Enumerate all counters after job is completed

```
for (CounterGroup group : counters) {
        System.out.println("* Counter Group: " + group.getDisplayName() + " (" +
        group.getName() + ")");
        System.out.println("  number of counters in this group: " + group.size());
        for (Counter counter : group) {
                System.out.println("  - " + counter.getDisplayName() + ": " +
                counter.getName() + ": "+counter.getValue());
        }
}
```

Counters in MRJob

- A counter has a group, a name, and an integer value. Hadoop itself tracks a few counters automatically. mrjob prints your job's counters to the command line when your job finishes, and they are available to the runner object if you invoke it programmatically.
- To increment a counter from anywhere in your job, use the increment_counter() method:

```
class MRCountingJob(MRJob):

    def steps(self):
        # 3 steps so we can check behavior of counters for multiple steps
        return [MRStep(self.mapper),
                MRStep(self.mapper),
                MRStep(self.mapper)]

    def mapper(self, _, value):
        self.increment_counter('group', 'counter_name', 1)
        yield _, value
```

- At the end of your job, you'll get the counter's total value.
- You can also read the counters by using "runner.counters()"

https://mrjob.readthedocs.io/en/latest/guides/runners.html

How to Find the Shortest Path?

- The parallel breadth-first search algorithm only finds the shortest distances.
- Store “back-pointers” at each node, as with Dijkstra's algorithm
  - Not efficient to recover the path from the back-pointers
- A simpler approach is to emit paths along with distances in the mapper, so that each node will have its shortest path easily accessible at all times
  - The additional space requirement is acceptable

BFS Pseudo-Code (Weighted Edges)

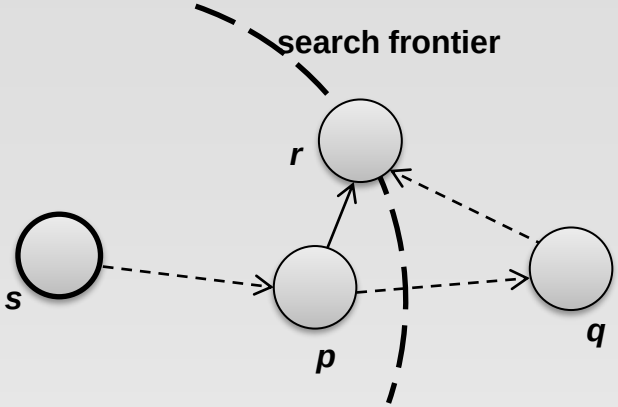
- The adjacency lists, which were previously lists of node ids, must now encode the edge distances as well
  - Positive weights!

- In line 6 of the mapper code, instead of emitting d + 1 as the value, we must now emit d + w, where w is the edge distance

- **The termination behaviour is very different!**
  - How many iterations are needed in parallel BFS (positive edge weight case)?
  - Convince yourself: when a node is first "discovered", we've found the shortest path

Not true!

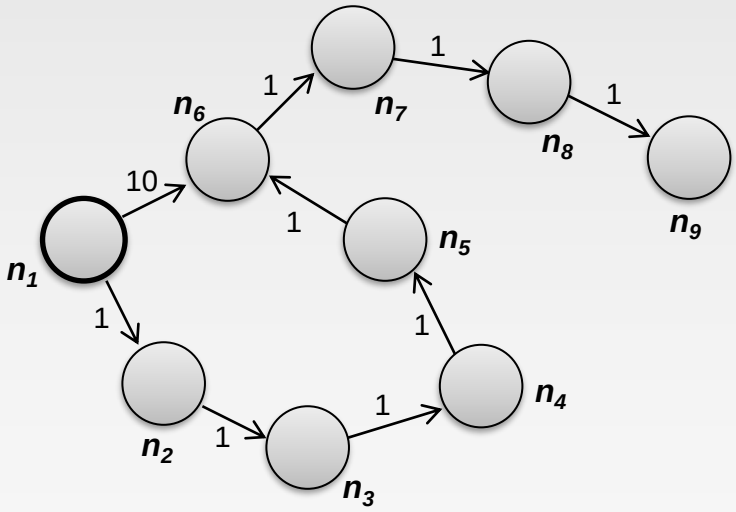
Additional Complexities



- Assume that *p* is the current processed node
  - In the current iteration, we just “discovered” node r for the very first time.
  - We've already discovered the shortest distance to node *p*, and that the shortest distance to *r* so far goes through *p*
  - Is *s->p->r* the shortest path from *s* to *r*?
- The shortest path from source *s* to node *r* may go outside the current search frontier
  - It is possible that *p->q->r* is shorter than *p->r*!
  - We will not find the shortest distance to *r* until the search frontier expands to cover *q*.

How Many Iterations Are Needed?

- In the worst case, we might need as many iterations as there are nodes in the graph minus one
  - A sample graph that elicits worst-case behaviour for parallel breadth-first search.
  - Eight iterations are required to discover shortest distances to all nodes from n_1.



Example (only distances)

- Input file:

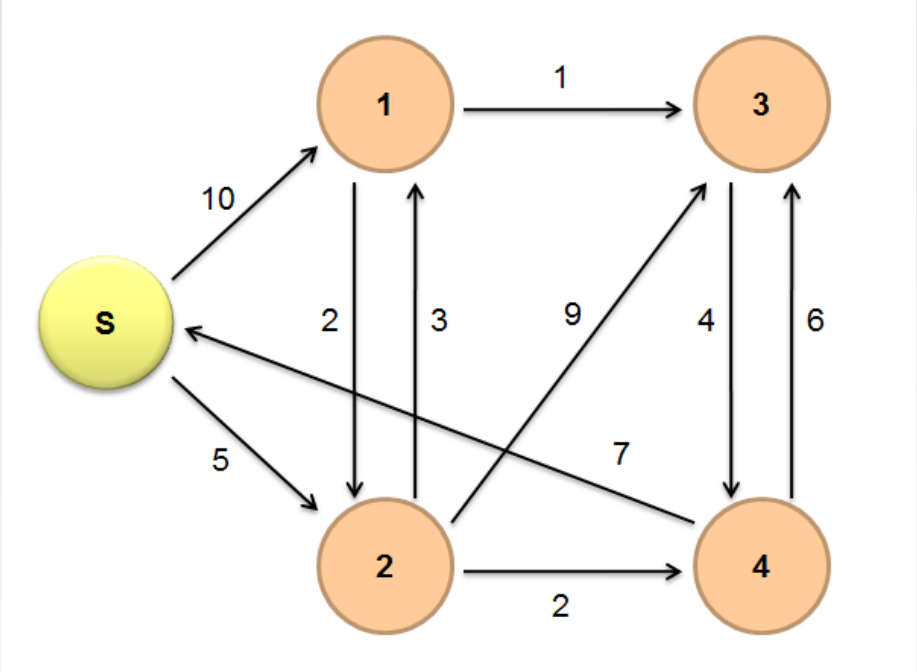
s --> 0 | n1: 10, n2: 5

n1 --> ∞ | n2: 2, n3:1

n2 --> ∞ | n1: 3, n3:9, n4:2

n3 --> ∞ | n4:4

n4 --> ∞ | s:7, n3:6



Iteration 1

- Map:

Read s --> 0 | n1: 10, n2: 5

Emit: (n1, 10), (n2, 5), and the adjacency list (s, n1: 10, n2: 5)

The other lists will also be read and emit, but they do not contribute, and thus ignored

- Reduce:

Receives: (n1, 10), (n2, 5), (s, <0, (n1: 10, n2: 5)>)

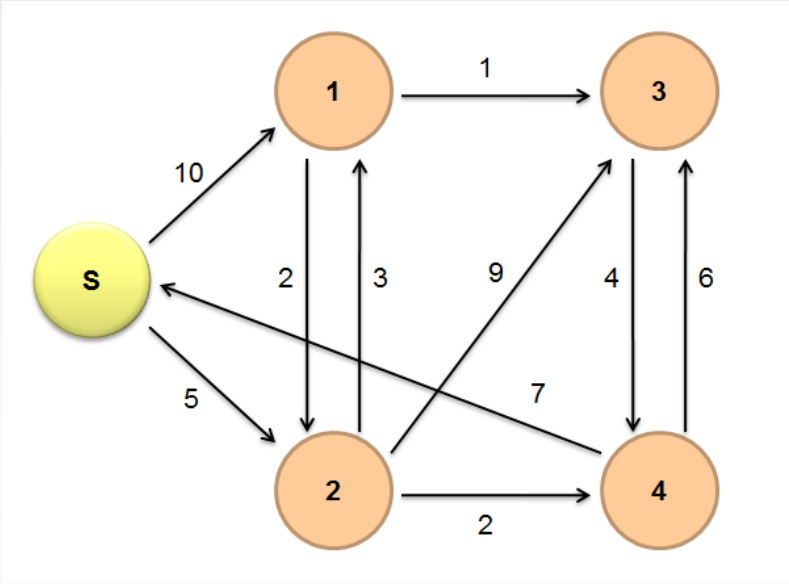
The adjacency list of each node will also be received, ignored in example

Emit:

s --> 0 | n1: 10, n2: 5

n1 --> 10 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2



Iteration 2

- Map:

Read: n1 --> 10 | n2: 2, n3:1

Emit: (n2, 12), (n3, 11), (n1, <10, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1: 3, n3:9, n4:2)>)

Ignore the processing of the other lists

- Reduce:

Receives: (n1, (8, <10, (n2: 2, n3:1)>)), (n2, (12, <5, n1: 3, n3:9, n4:2>)), (n3, (11, 14)), (n4, 7)

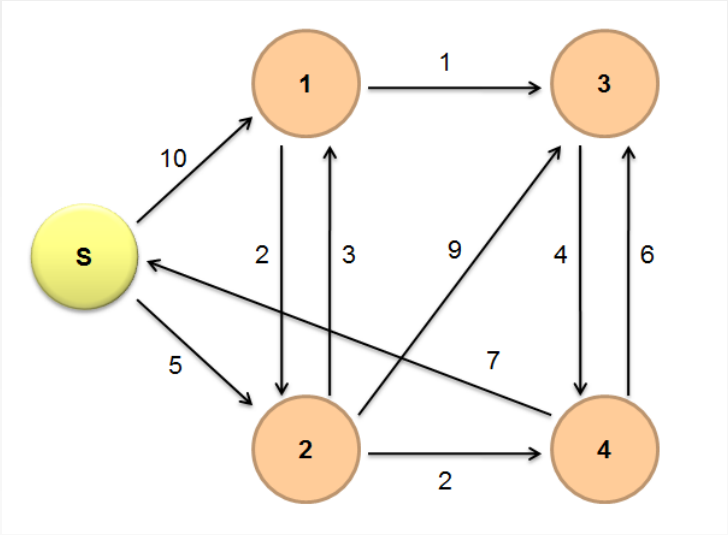
Emit:

n1 --> 8 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 11 | n4:4

n4 --> 7 | s:7, n3:6



Iteration 3

- Map:

Read: n1 --> 8 | n2: 2, n3:1

Emit: (n2, 10), (n3, 9), (n1, <8, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2 (**Again!**)

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1: 3, n3:9, n4:2)>)

Read: n3 --> 11 | n4:4

Emit: (n4, 15), (n3, <11, (n4:4)>)

Read: n4 --> 7 | s:7, n3:6

Emit: (s, 14), (n3, 13), (n4, <7, (s:7, n3:6)>)

- Reduce:

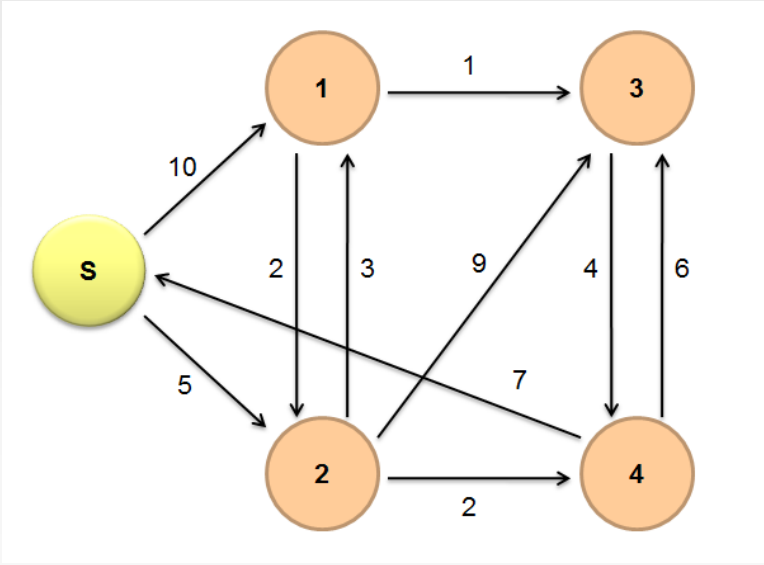
Emit:

n1 --> 8 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 9 | n4:4

n4 --> 7 | s:7, n3:6



Iteration 4

In order to avoid duplicated computations, you can use a status value to indicate whether the distance of the node has been modified in the previous iteration.

- Map:

Read: n1 --> 8 | n2: 2, n3:1 (**Again!**)

Emit: (n2, 10), (n3, 9), (n1, <8, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2 (**Again!**)

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1: 3, n3:9, n4:2)>)

Read: n3 --> 9 | n4:4

Emit: (n4, 13), (n3, <9, (n4:4)>)

Read: n4 --> 7 | s:7, n3:6 (**Again!**)

Emit: (s, 14), (n3, 13), (n4, <7, (s:7, n3:6)>)

- Reduce:

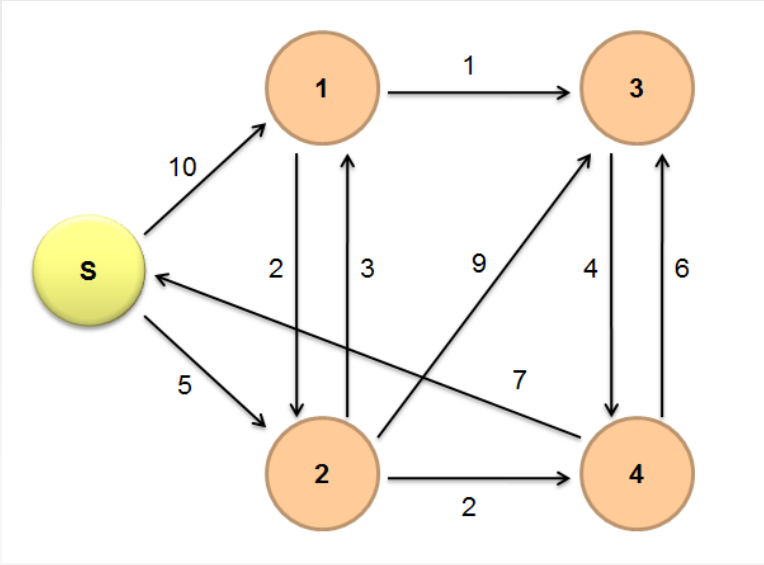
Emit:

n1 --> 8 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 9 | n4:4

n4 --> 7 | s:7, n3:6



No updates. Terminate.

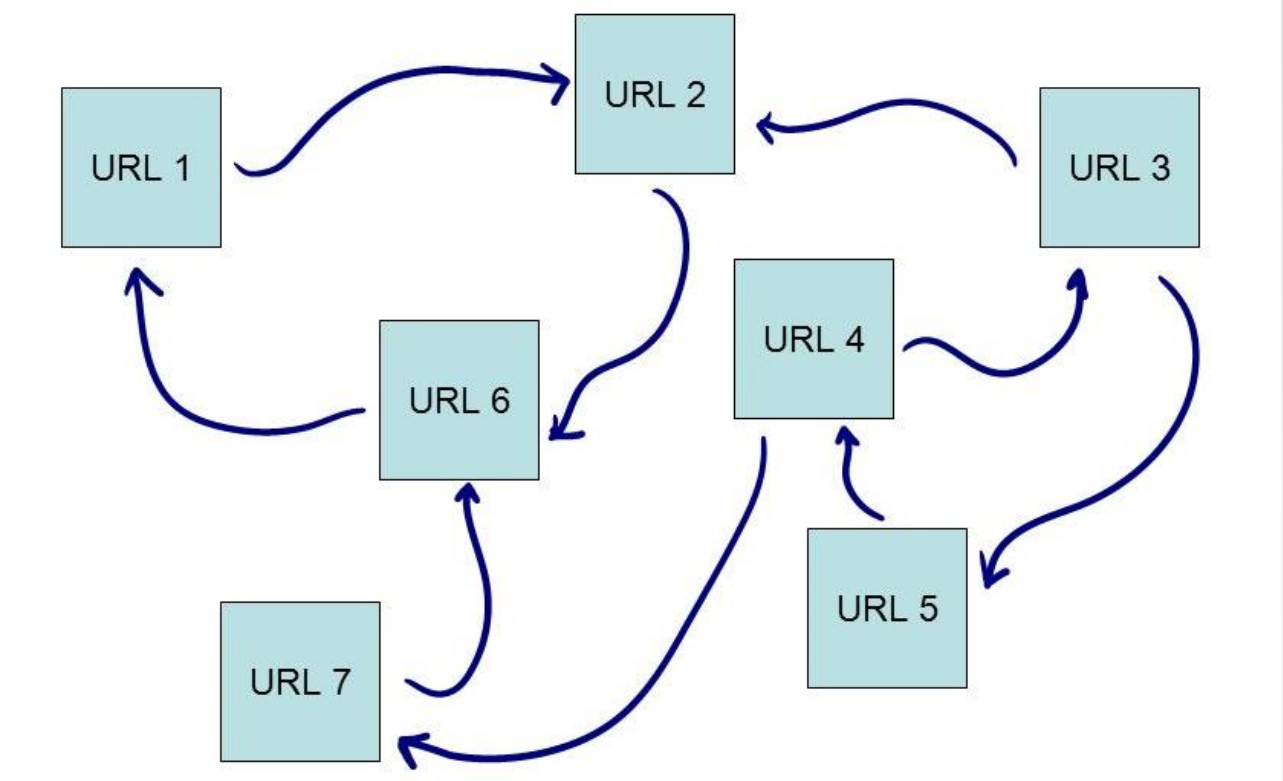
Comparison to Dijkstra

- Dijkstra's algorithm is more efficient
  - At any step it only pursues edges from the minimum-cost path inside the frontier
- MapReduce explores all paths in parallel
  - Lots of "waste"
  - Useful work is only done at the "frontier"
- Why can't we do better using MapReduce?

PageRank

Web as a Directed Graph

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



Broad Question

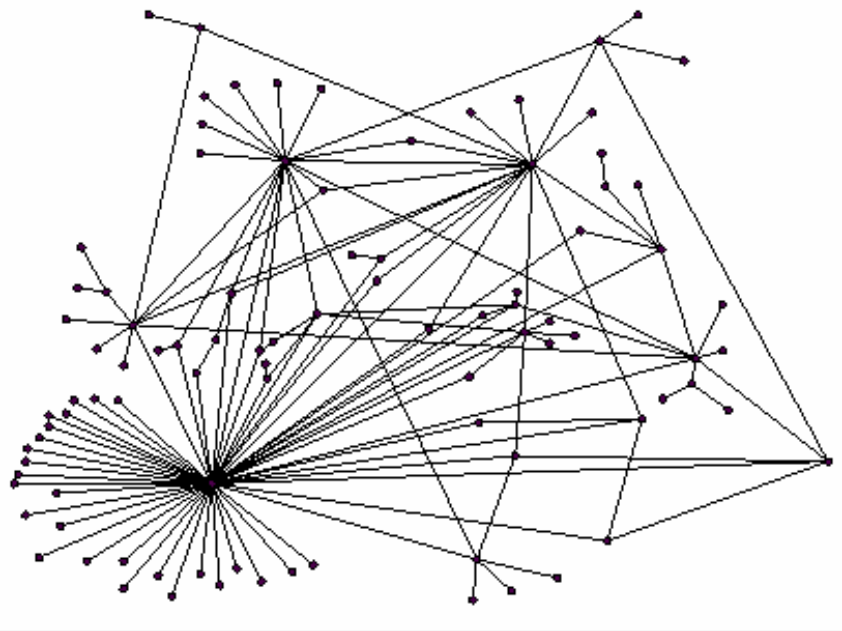
- How to organize the Web?
- First try: Human curated Web directories
  - Yahoo, LookSmart, etc.
- Second try: Web Search
  - Information Retrieval investigates: Find relevant docs in a small and trusted set
    - Newspaper articles, Patents, etc.
  - But: Web is huge, full of untrusted documents, random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

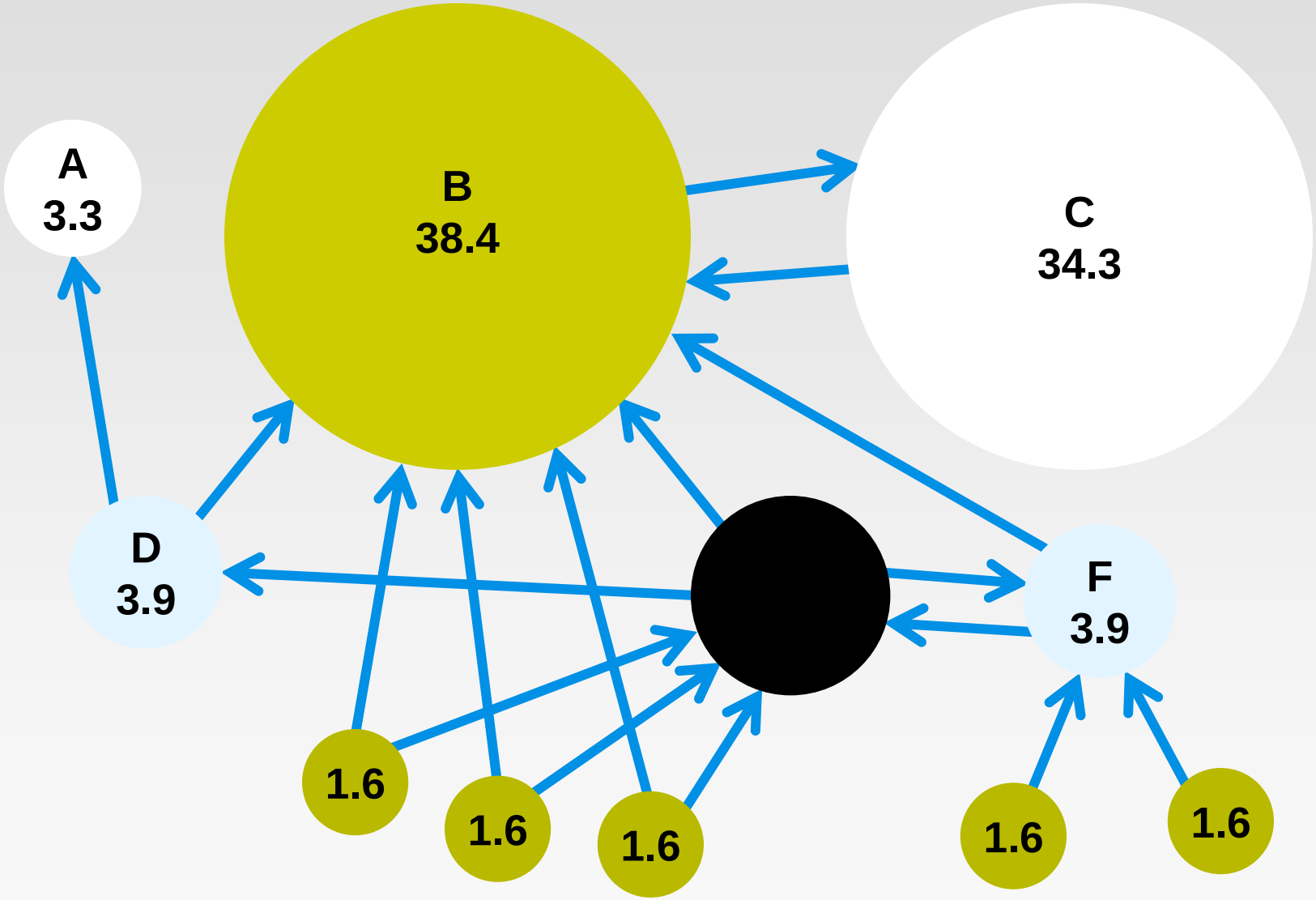
Links as Votes

- 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!

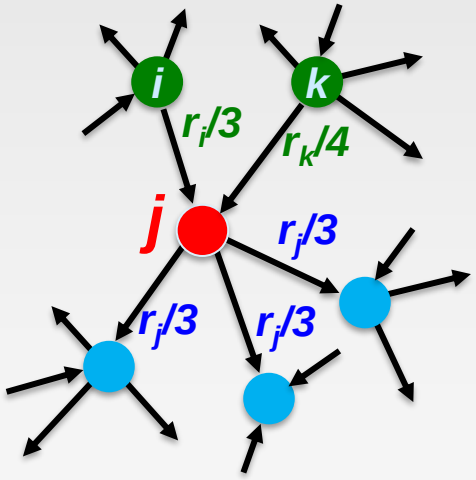
Example: PageRank Scores



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

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

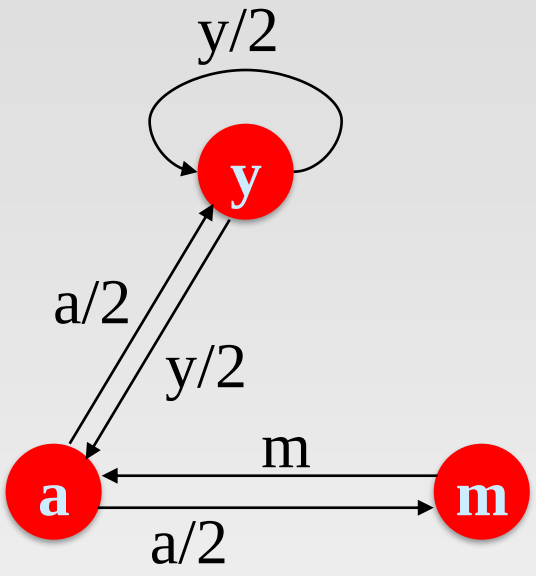


PageRank: The "Flow" Model

- 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

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

d_i … out-degree of node i



"Flow" equations:

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

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:
  - $r_y + r_a + r_m = 1$
  - **Solution:** $r_y = \frac{2}{5}$, $r_a = \frac{2}{5}$, $r_m = \frac{1}{5}$

- 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

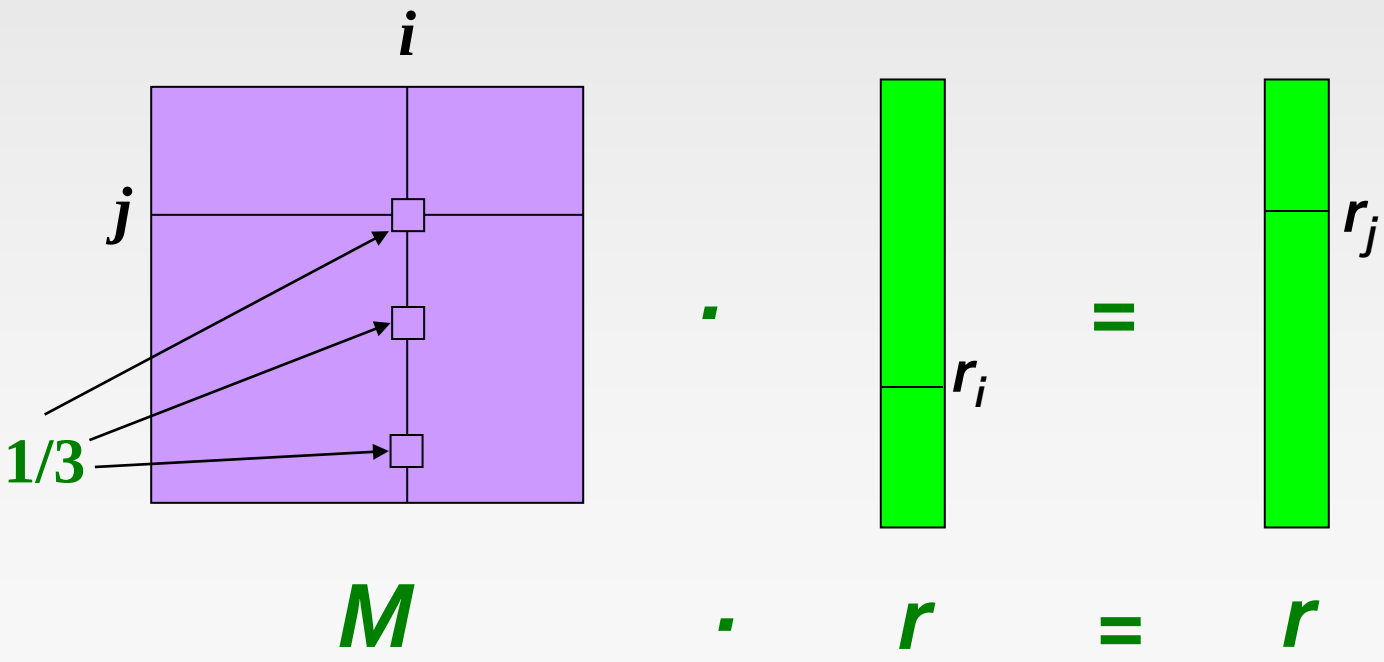
$$\boldsymbol{r} = \boldsymbol{M} \cdot \boldsymbol{r}$$

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

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

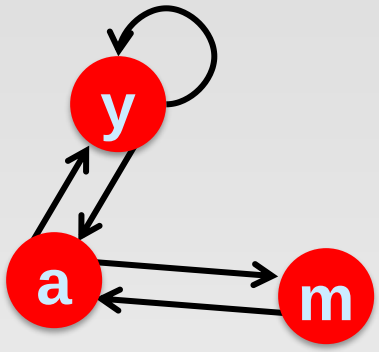
- So the **rank vector** ***r*** is an **eigenvector** of the stochastic web matrix ***M***
  - In fact, its first or principal eigenvector, with corresponding eigenvalue ***1***
    - Largest eigenvalue of ***M*** is **1** since ***M*** is column stochastic (with non-negative entries)
      - *We know **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

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

$$r = M \cdot r$$

$$\begin{bmatrix} y \\ a \\ m \end{bmatrix} = \begin{bmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{bmatrix} \begin{bmatrix} y \\ a \\ m \end{bmatrix}$$

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 $|\mathbf{r}^{(t+1)} - \mathbf{r}^{(t)}|_1 < \varepsilon$

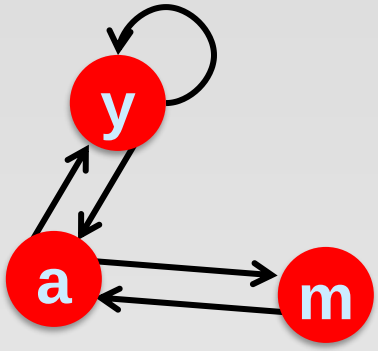
$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{\mathrm{d_i}}$$

d_i …. out-degree of node i

$|\mathbf{x}|_1 = \sum_{1 \le i \le N} |x_i|$ is the **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

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

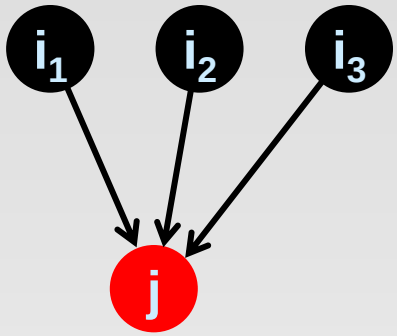
- **Example:**

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

Iteration 0, 1, 2, …

Random Walk Interpretation

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



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

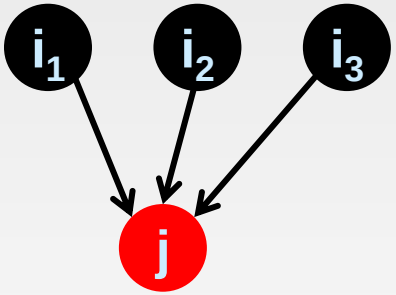
The Stationary Distribution

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

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

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

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

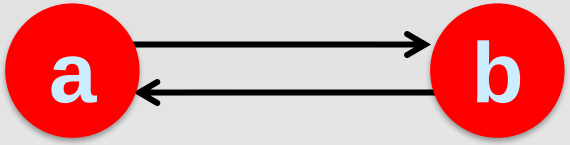
$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{\mathrm{d_i}}$$

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:

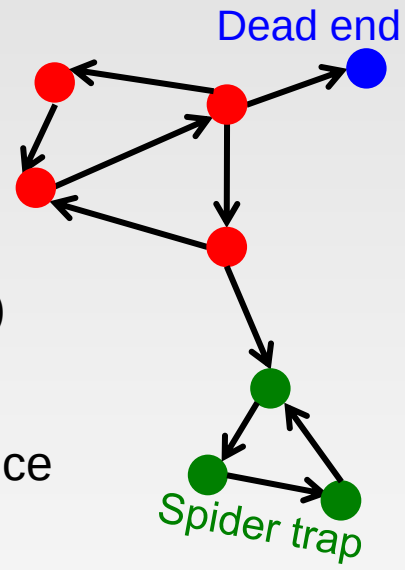
r_a	1	0	0	0
r_b	0	1	0	0

Iteration 0, 1, 2, …

PageRank: Problems

2 problems:

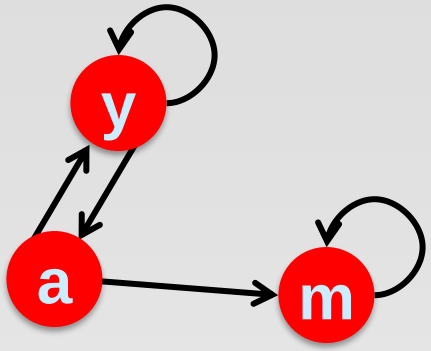
- **(1)** Some pages are **dead ends** (have no out-links)
  - Random walk has “nowhere” to go to
  - Such pages cause importance to “leak out”



- **(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:**

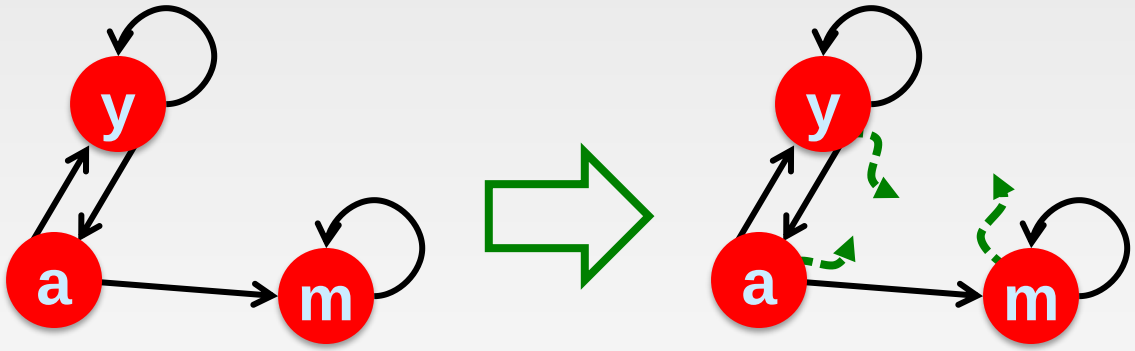
$$\begin{pmatrix} r_y \\ r_a \\ r_m \end{pmatrix} = \begin{matrix} 1/3 & 2/6 & 3/12 & 5/24 & & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \ldots & 0 \\ 1/3 & 3/6 & 7/12 & 16/24 & & 1 \end{matrix}$$

Iteration 0, 1, 2, …

All the PageRank score gets “trapped” in node m.

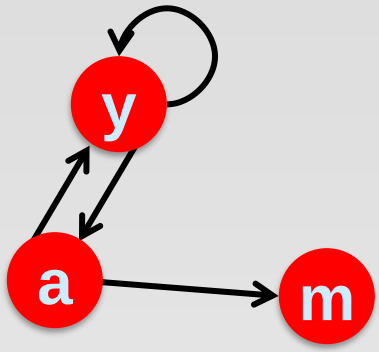
Solution: Teleport!

- 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

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

- **Example:**

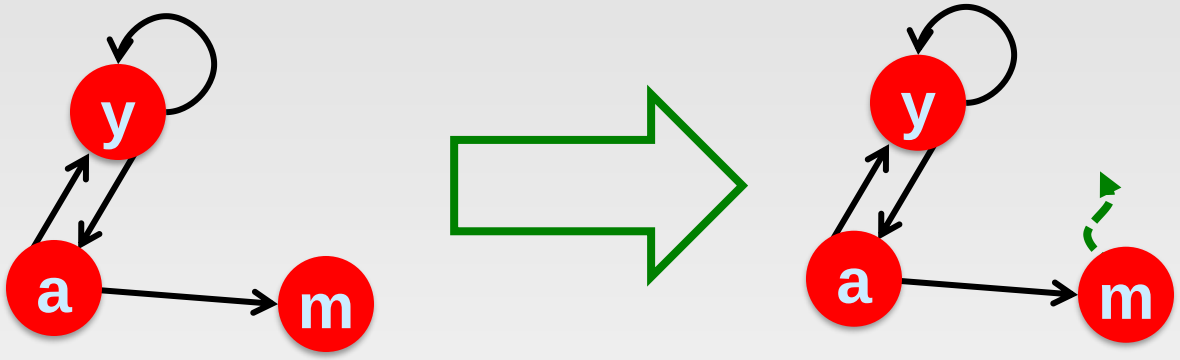
$$\begin{pmatrix} r_y \\ r_a \\ r_m \end{pmatrix} = \begin{matrix} 1/3 & 2/6 & 3/12 & 5/24 & & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \ldots & 0 \\ 1/3 & 1/6 & 1/12 & 2/24 & & 0 \end{matrix}$$

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 $\boldsymbol{M}$ has no dead ends. We can either preprocess matrix $\boldsymbol{M}$ 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]

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

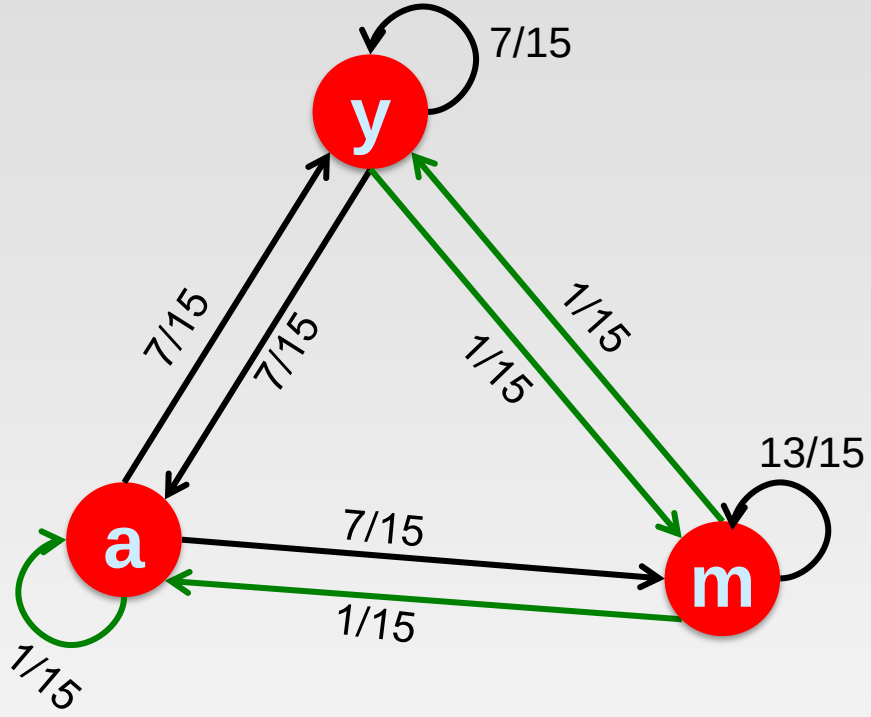
- **The Google Matrix *A*:**

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

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

- **We have a recursive problem: $r = A \cdot r$**
  And the Power method still works!
- **What is β?**
  - In practice $\beta = 0.8, 0.9$ (make 5 steps on avg., jump)

Random Teleports (β = 0.8)



M

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

$[1/N]_{NxN}$

$$\begin{array}{c} \\ y \\ a \\ m \end{array} \begin{bmatrix} 7/15 & 7/15 & 1/15 \\ 7/15 & 1/15 & 1/15 \\ 1/15 & 7/15 & 13/15 \end{bmatrix}$$

A

$$\begin{array}{c} y \\ a \\ m \end{array} = \begin{array}{c} 1/3 \\ 1/3 \\ 1/3 \end{array} \begin{array}{c} 0.33 \\ 0.20 \\ 0.46 \end{array} \begin{array}{c} 0.24 \\ 0.20 \\ 0.52 \end{array} \begin{array}{c} 0.26 \\ 0.18 \\ 0.56 \end{array} \ldots \begin{array}{c} 7/33 \\ 5/33 \\ 21/33 \end{array}$$

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!

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

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

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

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

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

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

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

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

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

PageRank: The Complete Algorithm

- **Input:** Graph G and parameter β
  - Directed graph G (can have **spider traps** and **dead ends**)
  - Parameter β
- **Output:** PageRank vector r^{new}

  - **Set:** $r_j^{old} = \frac{1}{N}$
  - **repeat until convergence:** $\sum_j \left| r_j^{new} - r_j^{old} \right| > \varepsilon$
    - $\forall j$: $r'^{new}_j = \sum_{i \to j} \beta \frac{r_i^{old}}{d_i}$

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

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

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

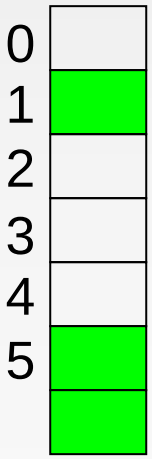
- Encode sparse matrix using only nonzero entries
  - Space proportional roughly to number of links
  - Say 10N, or 4*10*1 billion = 40GB
  - **Still won't fit in memory, but will fit on disk**

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:

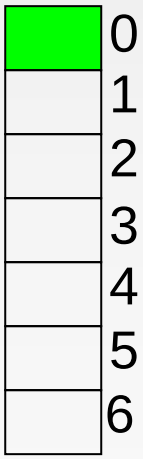
Initialize all entries of $r^{new} = (1-\beta) / N$
For each page i (of out-degree d_i):
  Read into memory: i, d_i, $dest_1, \ldots, dest_{d_i}$, $r^{old}(i)$
  For $j = 1 \ldots d_i$
    $r^{new}(dest_j) \mathrel{+}= \beta\, r^{old}(i) / d_i$



r^{new}

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

r^{old}



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

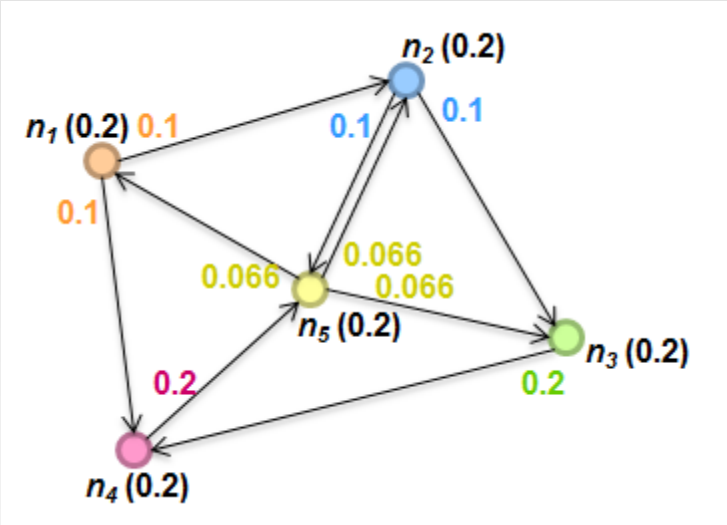
PageRank Computation Review

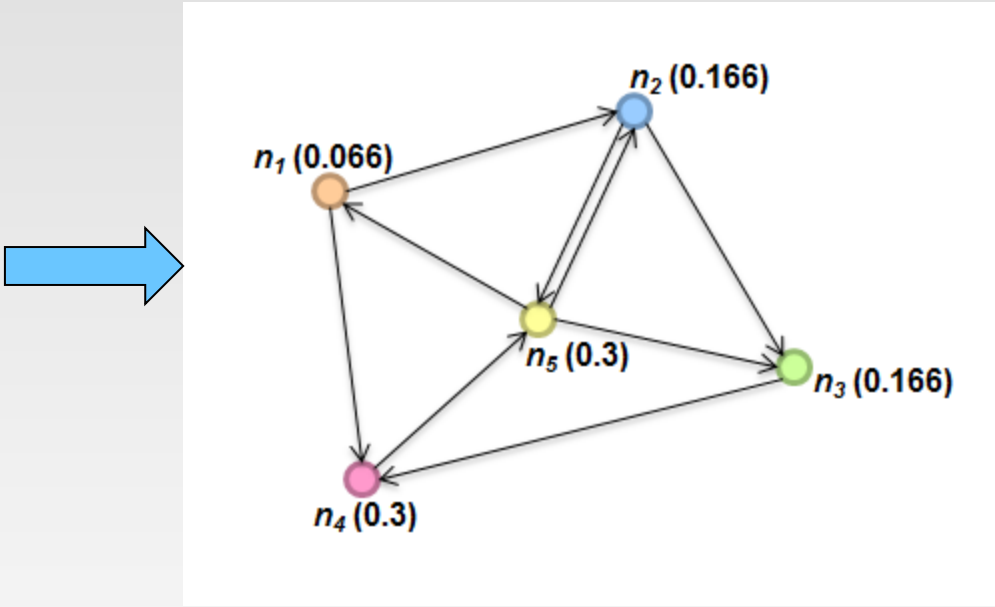
- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm:
  - Start with seed r_i values
  - Each page distributes r_i "credit" to all pages it links to
  - Each target page t_j adds up "credit" from multiple in-bound links to compute r_j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

Simplified PageRank

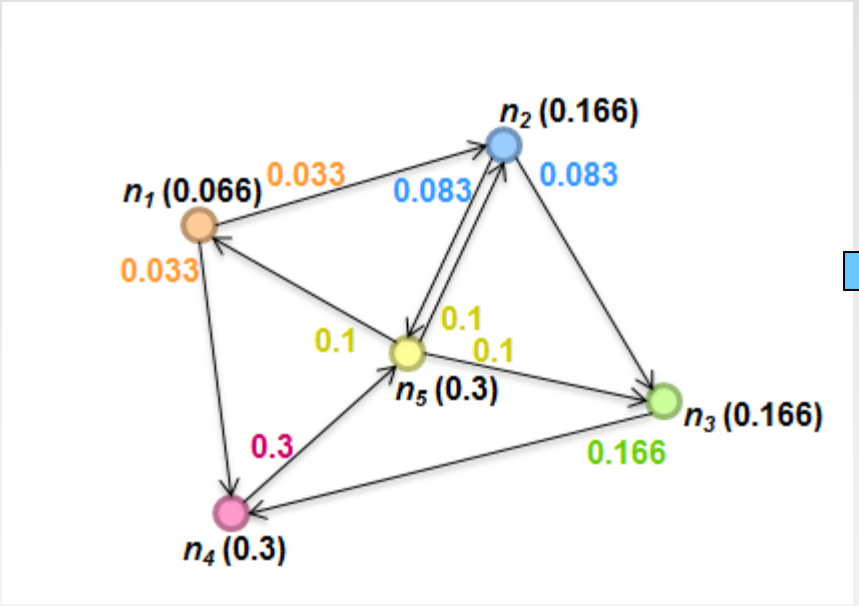
- First, tackle the simple case:
  - No teleport
  - No dead ends
- Then, factor in these complexities…
  - How to deal with the teleport probability?
  - How to deal with dead ends?

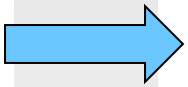
Sample PageRank Iteration (1)

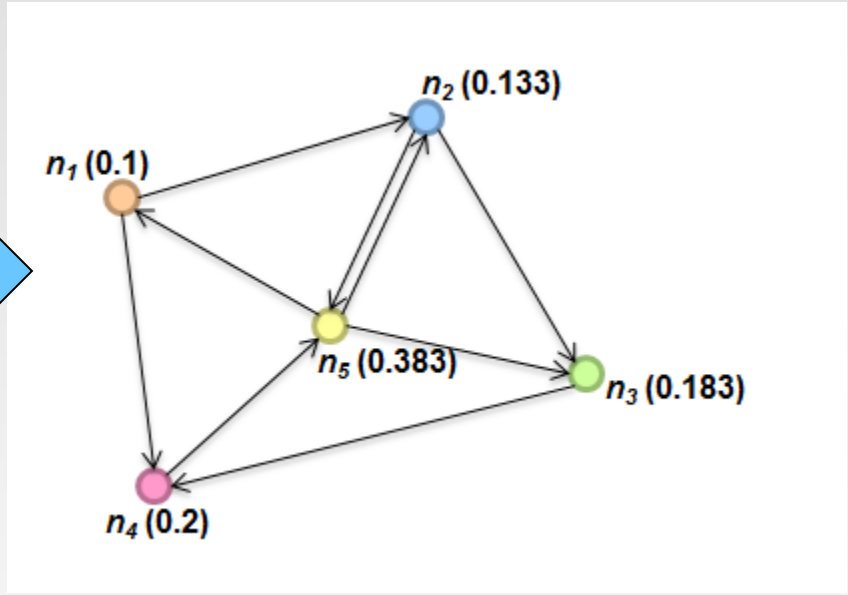




Sample PageRank Iteration (2)







PageRank in MapReduce

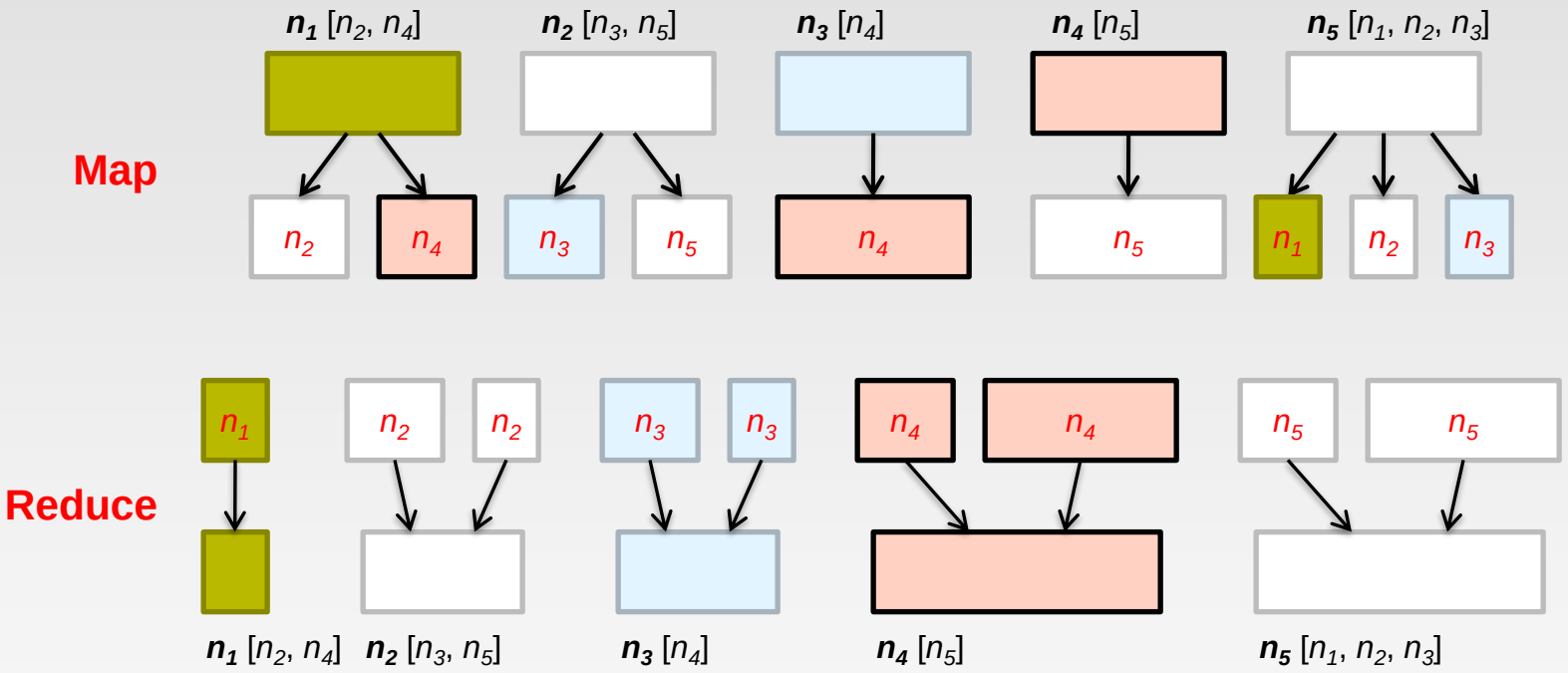
- One iteration of the PageRank algorithm involves taking an estimated PageRank vector r and computing the next estimate r′ by

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

- 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

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

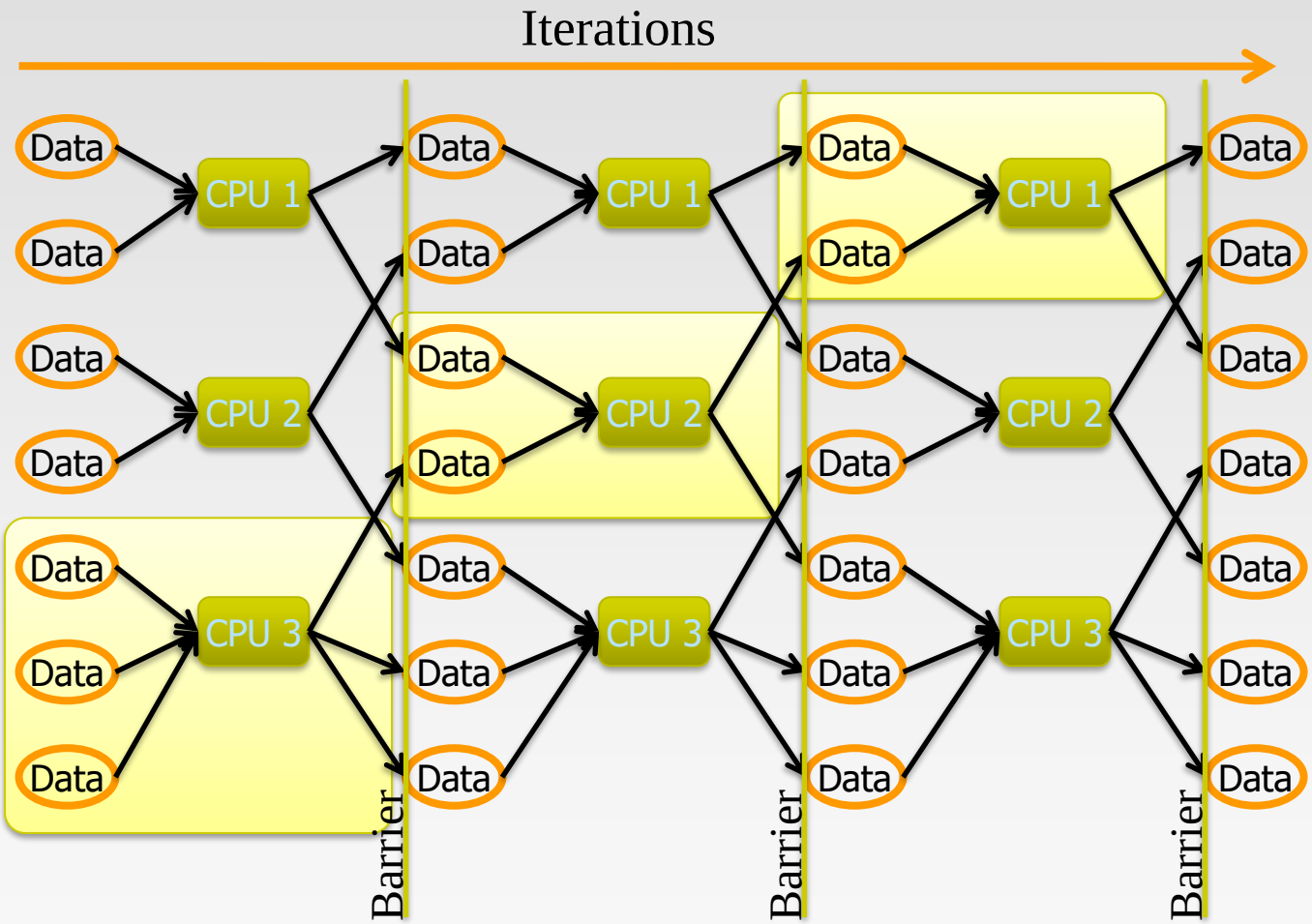
- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: “traversing” the graph
- Generic recipe:
  - Represent graphs as adjacency lists
  - Perform local computations in mapper
  - Pass 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
    - Dijkstra's algorithm on a single machine: a global priority queue that guides the expansion of nodes
    - Dijkstra‘s algorithm in Hadoop, no such queue available. Do some “wasted” computation instead
- 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.

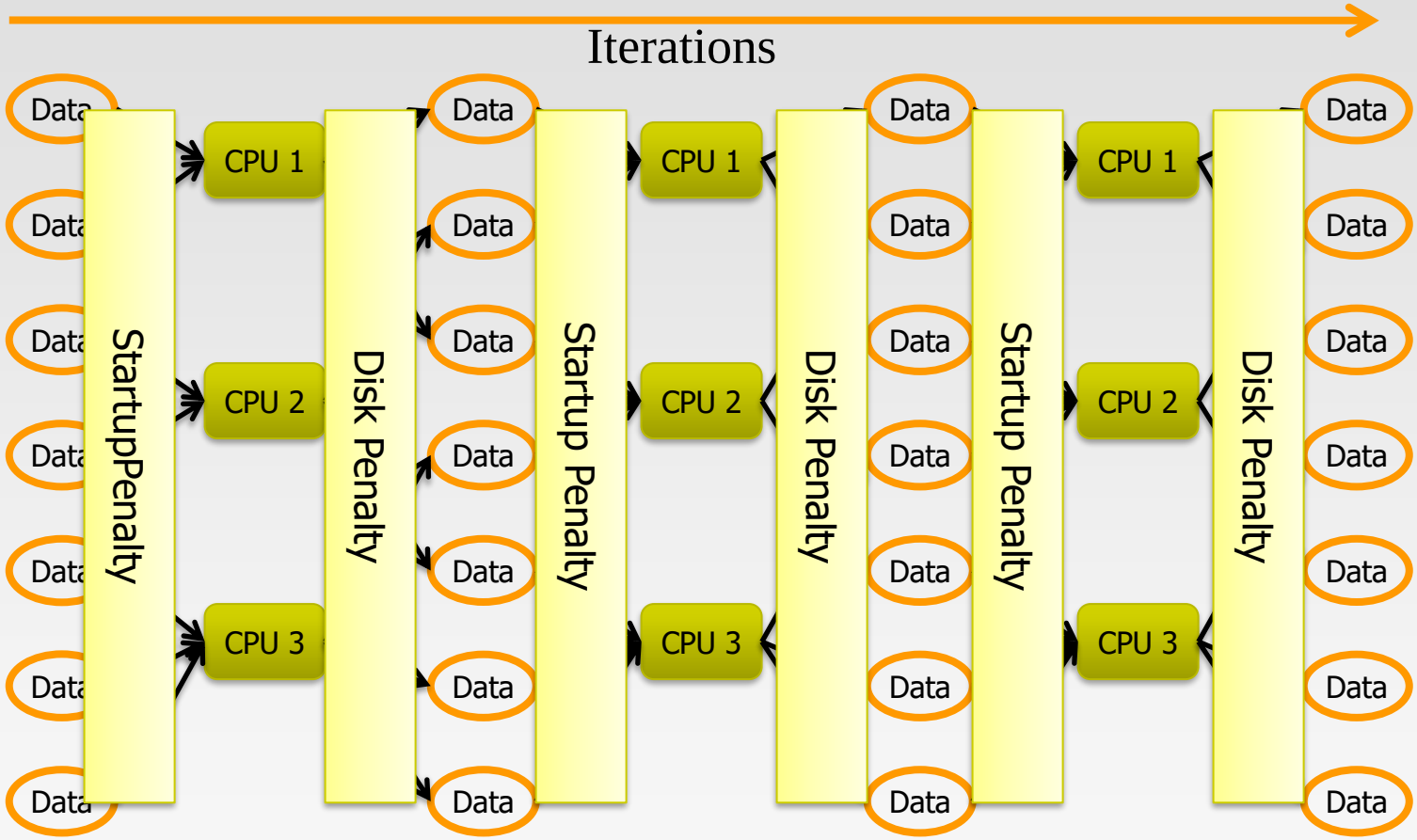
Iterative MapReduce

- 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

References

- Chapter 5, Data-Intensive Text Processing with MapReduce. Jimmy Lin and Chris Dyer. University of Maryland, College Park.

End of Chapter 8.1