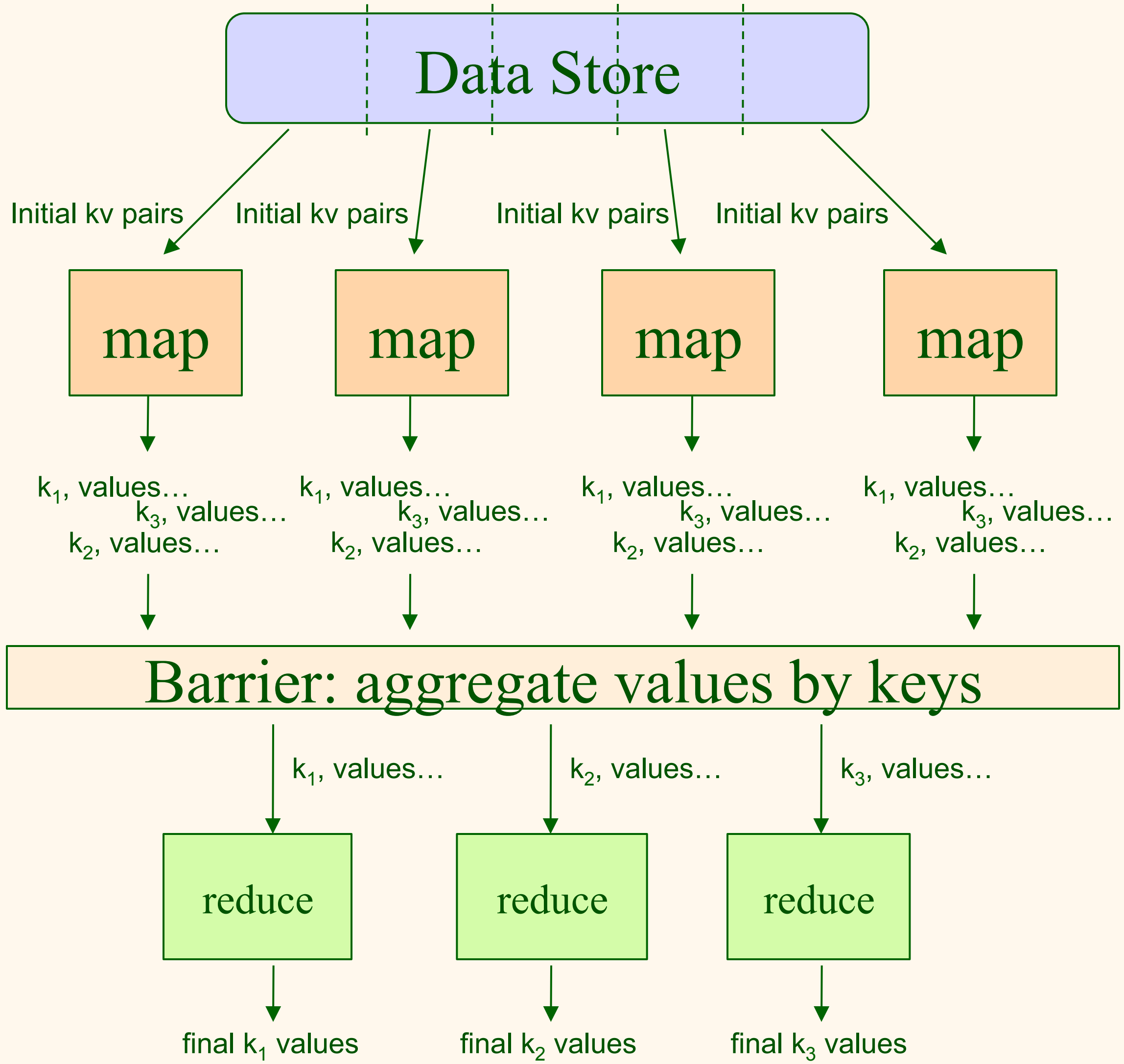
Map Reduce Continued

Example: Word Count

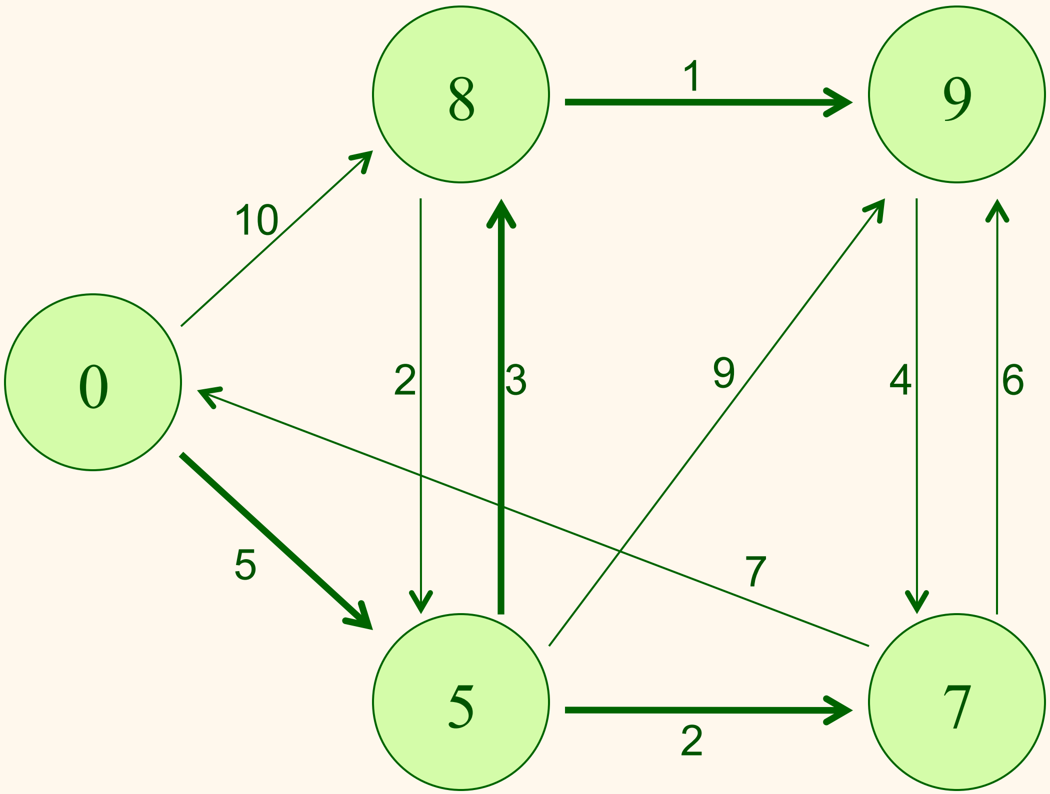
```
map(String key, String value):
      // key: document id
      // value: document contents
      for each word w in value:
            EmitIntermediate(w, "1");

reduce(String key, Iterator values):
      // key: a word
      // values: a list of counts
      int result = 0;
      for each v in values:
            result += ParseInt(v);
      Emit(key, AsString(result));
```

Map Reduce


Data Store
Initial kv pairs
Initial kv pairs
Initial kv pairs
Initial kv pairs
map
map
map
map
k_1, values…
k_3, values…
k_2, values…
k_1, values…
k_3, values…
k_2, values…
k_1, values…
k_3, values…
k_2, values…
k_1, values…
k_3, values…
k_2, values…
Barrier: aggregate values by keys
k_1, values…
k_2, values…
k_3, values…
reduce
reduce
reduce
final k_1 values
final k_2 values
final k_3 values

SSSP - Dijkstra's Algorithm

Single-Source-Shortest-Path



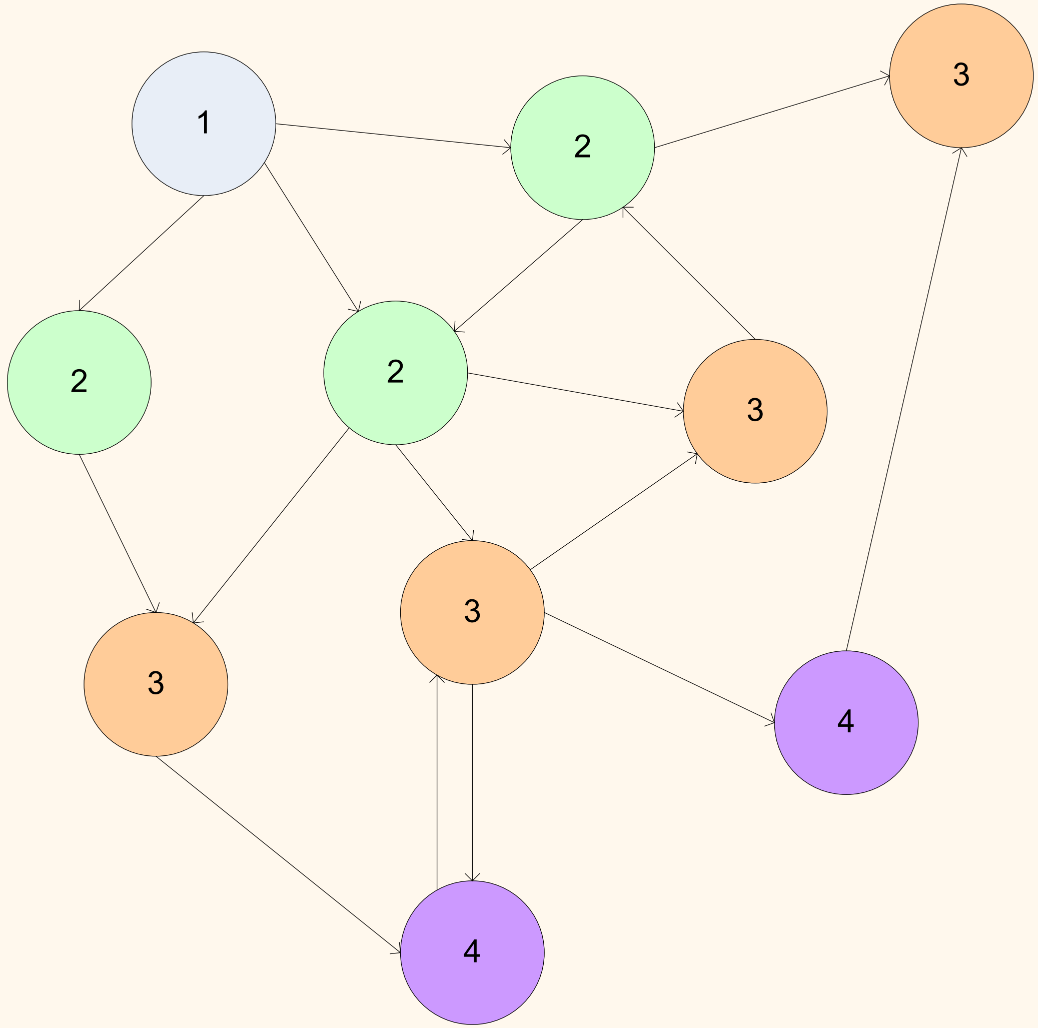
SSSP in Map Reduce

- Can't run Dijkstra's algorithm directly
  - Can't have a global queue!
- Another way to do it: Parallel BFS
- Start by assuming all edge weights are equal
  - Will relax this later

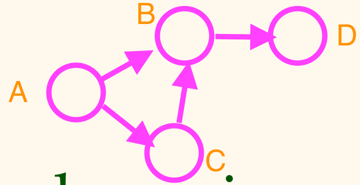
Finding the Shortest Path

- Intuition: process all nodes at each step
- Some nodes have no information (distance = infinity)
  - So can't do much
- But other nodes do know something
  - E.g. source knows it is at distance 0
  - So can pass this fact on to its out-neighbors
    - Who now know they are at distance 1!
  - At the next iteration, these neighbors know they're at distance 1
    - So can tell *their* out-neighbors they're at distance 2.

Parallel BFS



From Intuition to Algorithm



Map	Reduce
(B,1)	(B,[1])
(C,1)	(C,[1])
(D,2)	(B,[1,2])
(B,2)	(C, [1])
(B,1)	(D, [1])
(C,1)	(D,[2])

- A map task receives
  - Key: node *n*
  - Value: D (distance from start), points-to (list of nodes reachable from *n*)
- $\forall p \in$ points-to: emit (p, D+1)
- The reduce task gathers possible distances to a given *p* and selects the minimum one

- Possible through the magic of the "sort and shuffle" between Map and Reduce
  - Map processes node and updates distances of out-neighbors
  - Reduce processes node based on info from its in-neighbors

Multiple Iterations Needed

- Each Map Reduce task advances the “known frontier” by one hop
  - Subsequent iterations include more reachable nodes as frontier advances
  - Multiple iterations are needed to explore entire graph
  - Feed output back into the same MapReduce task

Multiple Iterations Needed

- Passing along the graph structure:
  - Next iteration of Map needs points-to list again
  - So need to "carry" it with us as we run the algorithm

Map

```
class Mapper
    method Map(nid n, node N)
        d ← N.Distance
        Emit(nid n, N)
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, d + 1)
```

Reduce

```
class Reducer
    method Reduce(nid m, [d1, d2, . . .])
        d_min ← ∞
        M ← Ø
        for all d ∈ counts [d1, d2, . . .] do
            if IsNode(d) then
                M ← d
            else if d < d_min then
                d_min ← d
        M.Distance← d_min
        Emit(nid m, node M)
```

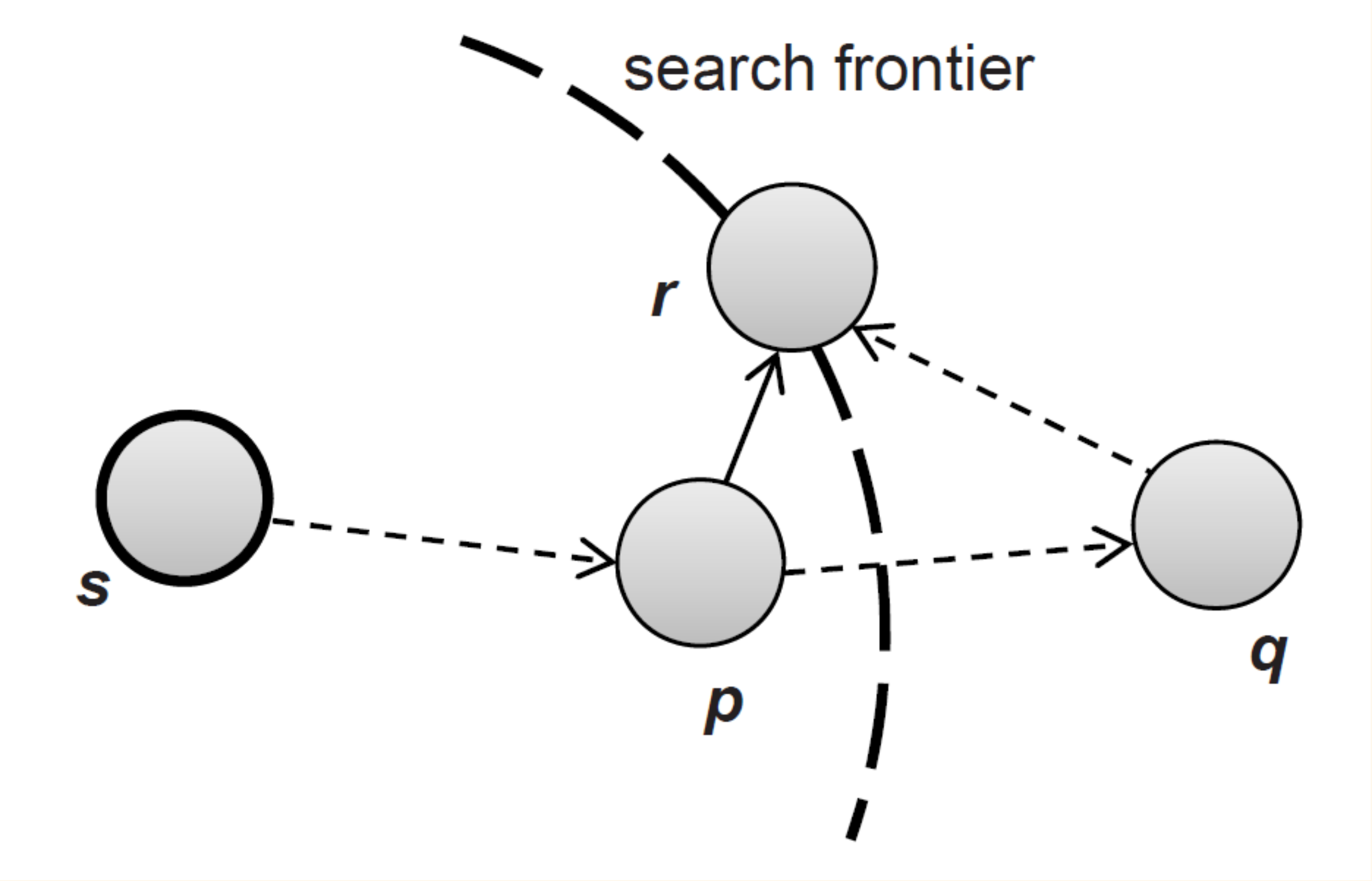
Termination

- Eventually, all nodes will be discovered, all edges will be considered (in a connected graph)
- Stop when there are no nodes with a distance of infinity
  - Can be checked by the driver/harness/program that runs the outer loop and schedules each Map Reduce job

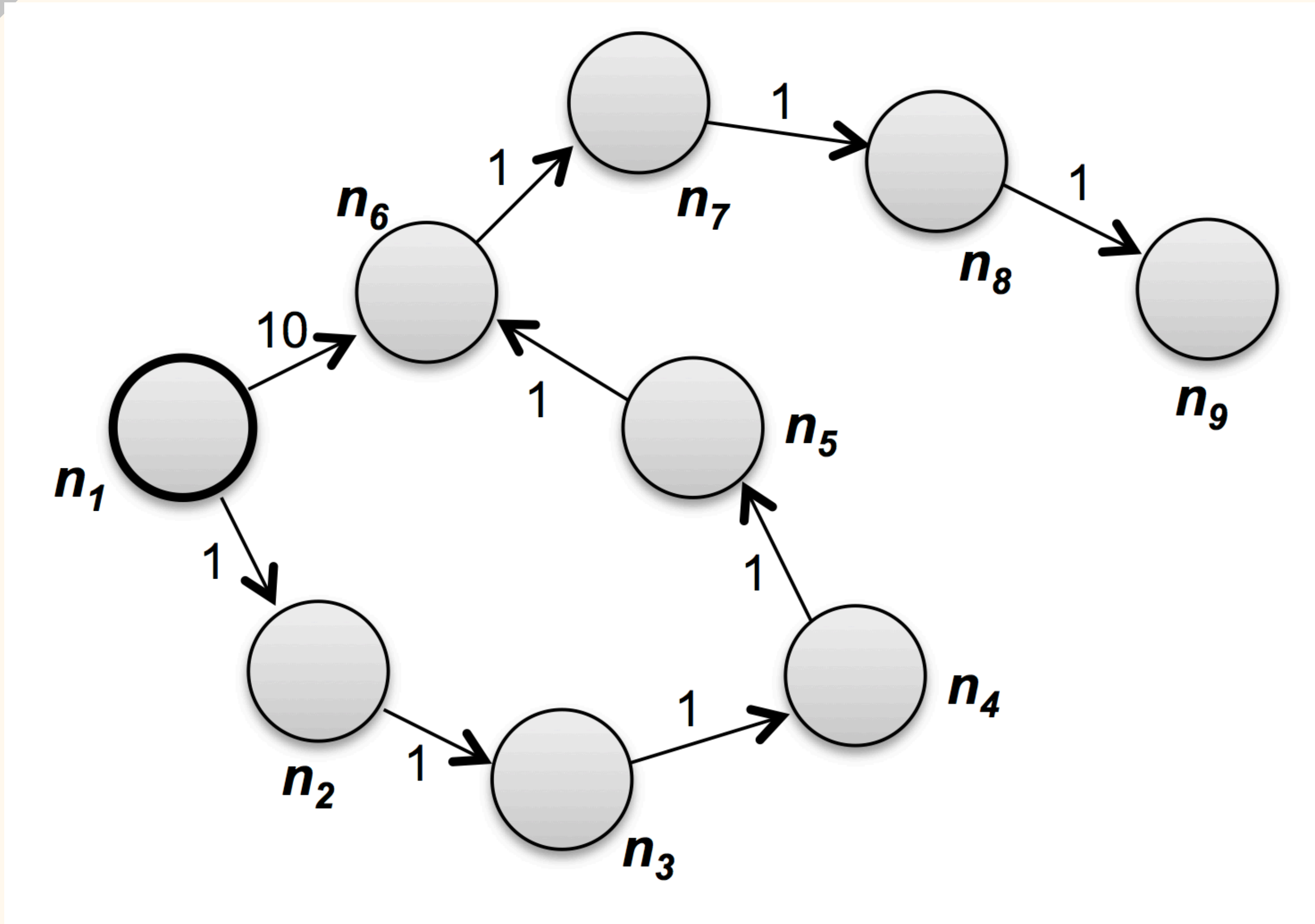
Weighted Edges

- Now add positive weights to the edges
- Simple change: points-to list in map task includes a weight *w* for each pointed-to node
  - emit (p, D+w_p) instead of (p, D+1) for each node p
- Termination behavior different
  - Just because we've reached a node doesn't mean we've found the shortest path to it!

Node Exploration Process



Node Exploration Process



Termination

- When distances have not changed during an iteration, safe to stop

Comparison to Dijkstra

- Dijkstra's algorithm is more efficient
  - At any step it only pursues edges from the minimum-cost path inside the frontier
  - Only processes each node once
- MapReduce explores all paths in parallel
  - Does a lot of recomputation
    - Not a bug, need it to handle situations where the "shortest" path contains more edges than another available path
  - But can be done in parallel

General Approach

- Graph algorithms with MapReduce:
  - Each map task receives a node and its outlinks
  - Map task compute some function of the link structure, emits value with target as the key
  - Reduce task collects keys (target nodes) and aggregates
- Iterate multiple MapReduce cycles until some termination condition

PageRank

- Google's famous algorithm for ranking Web Pages
  - A measure of "quality reputation" of a page
  - Useful for ranking/ordering search results

PageRank

- Based on link structure (graph) of the web
- Intuition from academic citation network:
  - Lots of citations (incoming links) = probably a high quality paper
  - If a high quality paper cites your paper, your paper is probably of high quality too
  - Vice versa not necessarily true (just by citing a high quality paper you don't make your paper high quality ☺)

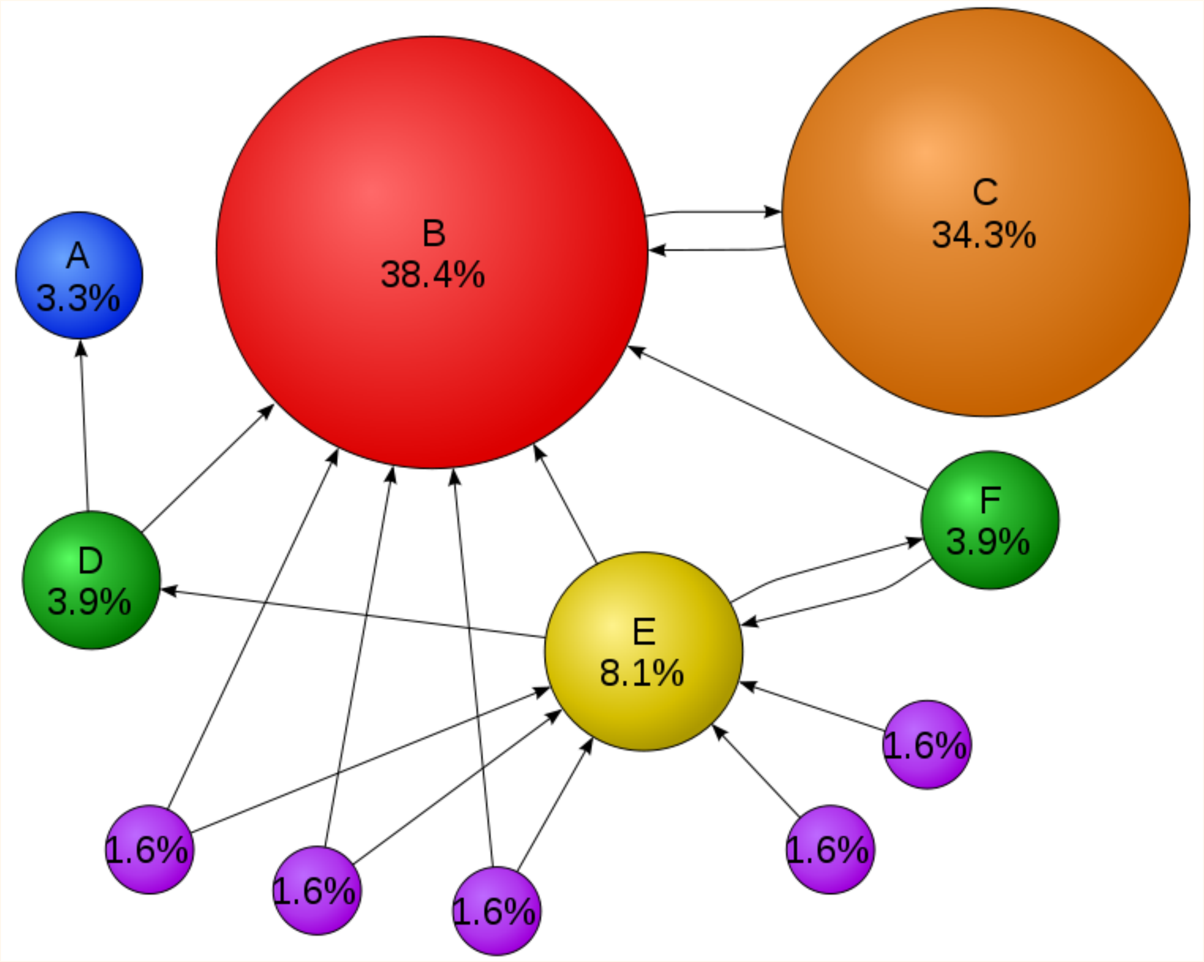
Random Surfer Model

- Another intuition for PageRank
- Imagine a surfer who starts on a randomly chosen page and then follows outgoing links at random
  - Markov process
- PageRank is probability that user will arrive at a given page during this random walk

A little more complex!

- Model assumes that surfer doesn't always follow a link, but sometimes e.g. bookmarks instead.
- Before each move, surfer flips a coin
  - With probability $1-\alpha$, follows an out-link
  - With probability α , teleports to a (uniformly chosen) random page

PageRank Example

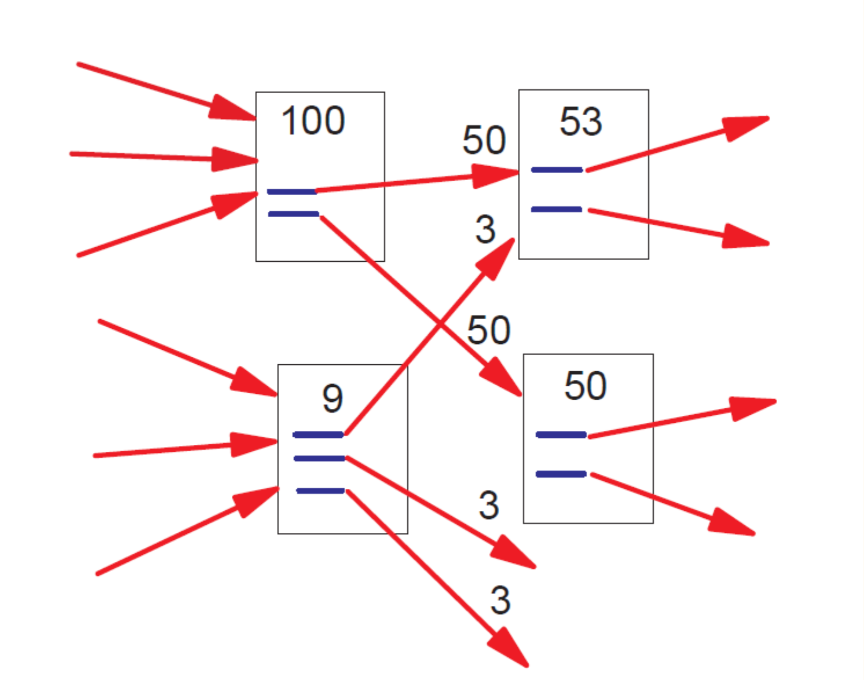


Computing PageRank

- Let's start with some simplifying assumptions
  - Assume all nodes have at least one outlink
  - And surfer never does a random restart using bookmarks
- Will talk about how to lift these assumptions soon

Simplified PageRank Intuition

- PageRank of a page is based on the PageRank of the pages which link to it
- A page divides its PageRank equally among all its outgoing links



Somewhat more formally

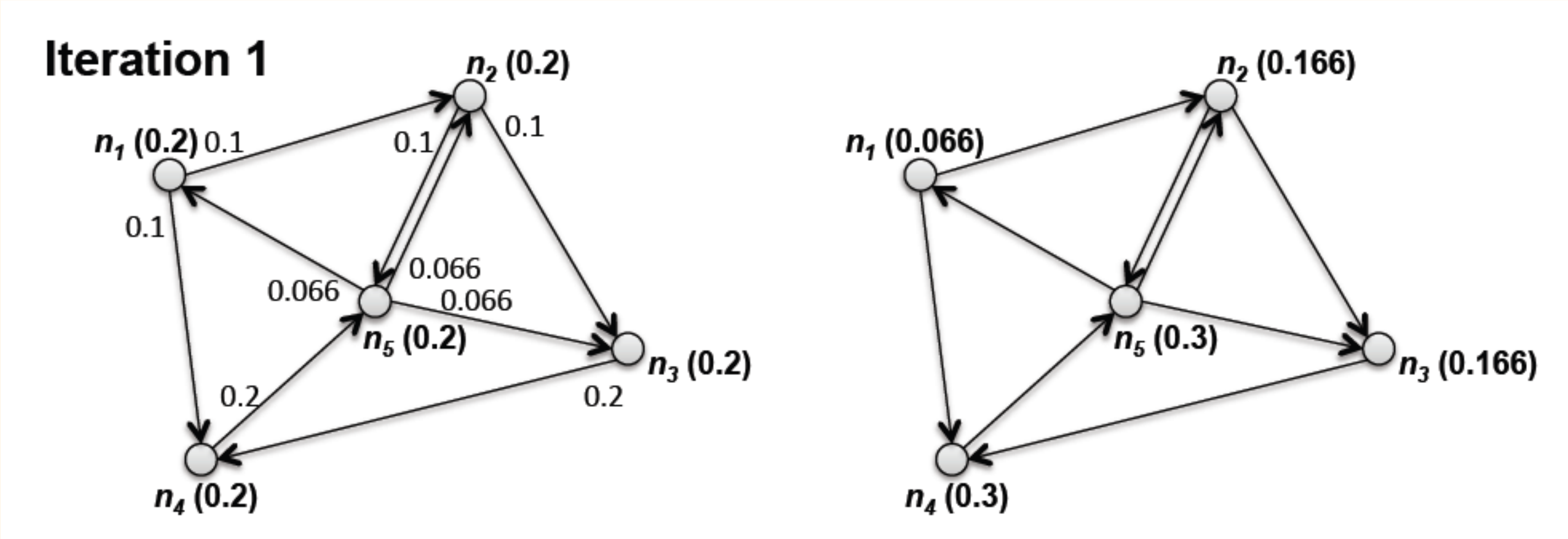
$$P(n) = \Sigma_{m \in L(n)} \frac{P(m)}{C(m)}$$

- L(n) is the set of pages that link to n and C(m) is the number of out-neigbors of page m

Iterative computation of SPR

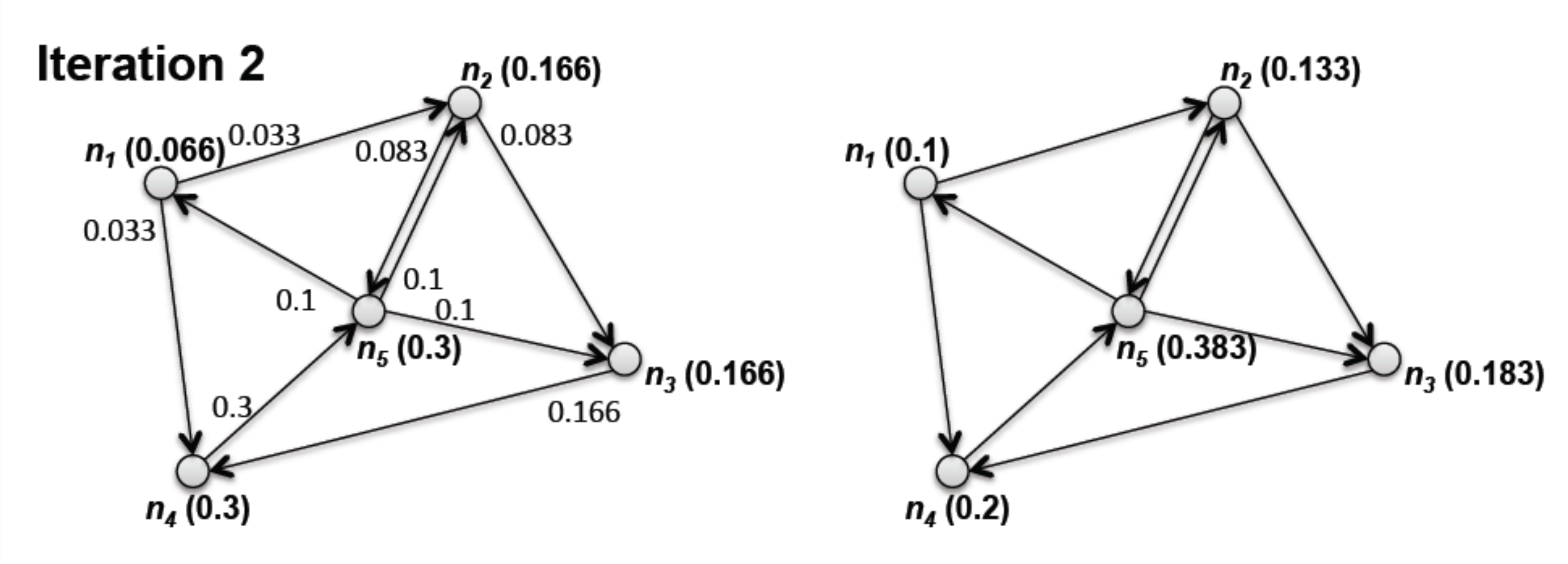
- How do we compute this?
- Various methods, but we are interested in Map Reduce
- Idea:
  - Initialize everything to the same PageRank (1/ number of nodes)
  - "pass around" PageRank contributions from nodes to their out-neighbors

Example



Iteration 1
n_1 (0.2)
n_2 (0.2)
n_3 (0.2)
n_4 (0.2)
n_5 (0.2)
0.1
0.1
0.1
0.1
0.066
0.066
0.066
0.2
0.2
n_1 (0.066)
n_2 (0.166)
n_3 (0.166)
n_4 (0.3)
n_5 (0.3)

Example


Iteration 2
n2 (0.166)
n1 (0.066)
0.033
0.083
0.083
0.033
0.1
0.1
0.1
n5 (0.3)
n3 (0.166)
0.3
0.166
n4 (0.3)
n2 (0.133)
n1 (0.1)
n5 (0.383)
n3 (0.183)
n4 (0.2)

Map

```
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
```

Reduce

```
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                          ▷ Sum incoming PageRank contributions
        M.PAGERANK ← s
        EMIT(nid m, node M)
```

Iterate

- Full algorithm is iterative
- Initialize the nodes to uniform distribution
- Run the two MR jobs described iteratively
- Until convergence (no change)

Now, back to dangling nodes

- If any PageRank is lost due to nodes with no out-neighbors, redistribute that PageRank uniformly throughout the graph for next iteration
- In the Map Reduce model: keep track of any lost PageRank
  - E.g. by using a special reserved intermediate key, or using a counter (i.e. storing it somewhere)

Solution

- After the MR task is done, do a cleanup pass
- Deal both with "missing mass" and with random restart factor

Cleanup pass

m:missing
G:number of all nodes

- Adjust the PageRank of each node to be

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

- Where *p* is the current PageRank, *m* is the mass lost due to sinks, and $|G|$ is the number of nodes in the graph
- This can be done using a map job (no reduce)

The full PageRank Algorithm

- Initialize the nodes to uniform distribution
- Run the two MR jobs described iteratively
- Until convergence (no change)

You can now start H4

- Available in CMS
- PageRank in Hadoop
  - And a second part on Neo4j (covered after break)

- Installation highly nontrivial
  - Consider starting over Spring Break if you are not experienced with command line Linux (or OS X)