

# Session 5: MapReduce – Graphs

Base on slides from Jimmy Lin



This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States  
See <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> for details

# Today's Agenda

- Graph problems and representations
- Parallel breadth-first search
- PageRank
- Beyond PageRank and other graph algorithms
- Optimizing graph algorithms

# 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 - transportation
  - Social networks (and electronic social networks)

# Some Graph Problems

- Finding shortest paths
  - Routing Internet traffic and UPS trucks
- Finding minimum spanning trees
  - Telco laying down fiber
- Finding Max Flow
  - Airline scheduling
- Identify “special” nodes and communities
  - Breaking up terrorist cells, spread of avian flu
- Bipartite matching
  - Monster.com, Match.com
- And of course... PageRank

# Graphs and MapReduce

- A large class of graph algorithms 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

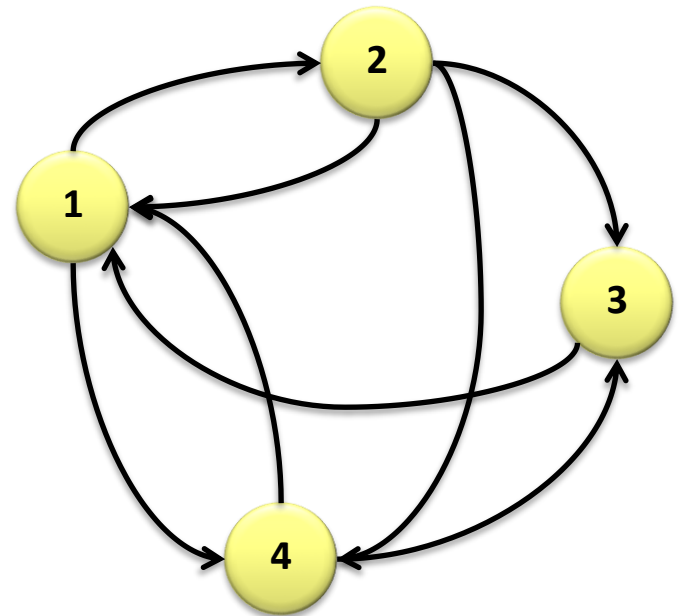
- $G = (V, E)$
- Two common representations
  - Adjacency matrix
  - Adjacency list

# Adjacency Matrices

Represent a graph as an  $n \times n$  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:

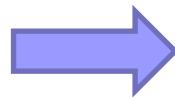
- Lots of zeros for sparse matrices
- Lots of wasted space



# 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

# Single-Source Shortest Path

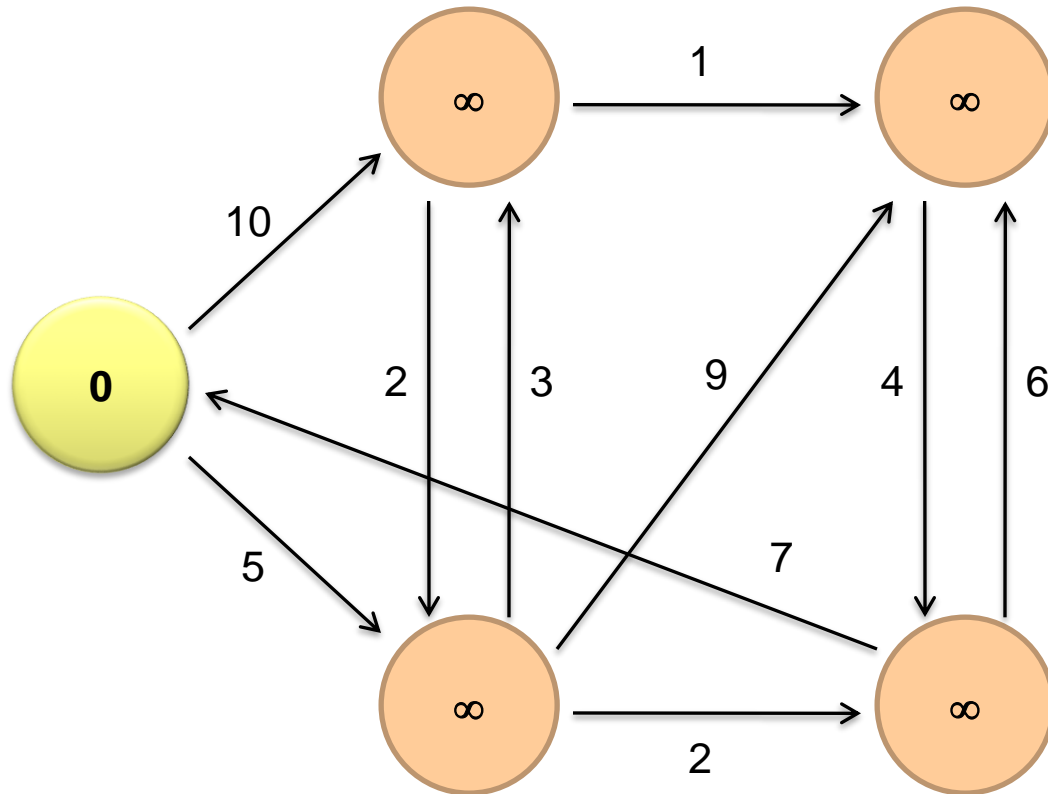
- **Problem:** find shortest path from a source node to one or more target nodes

- Shortest might also mean lowest weight or cost

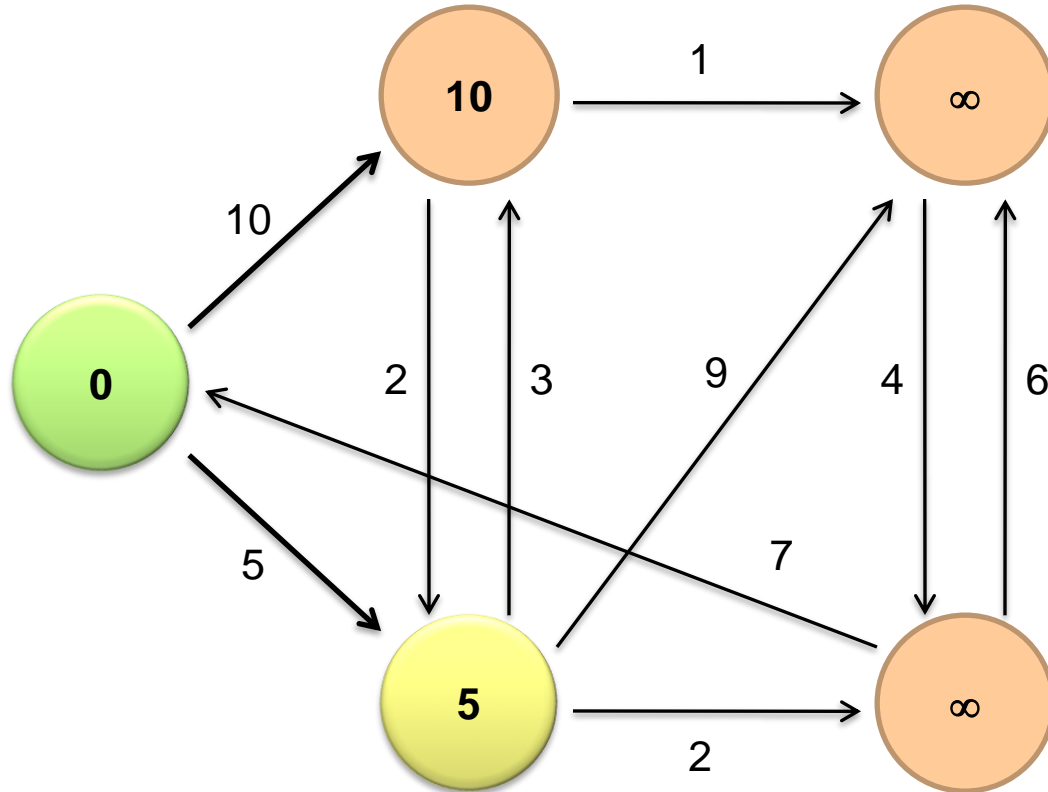
- First, a refresher: Dijkstra's Algorithm

```
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)
```

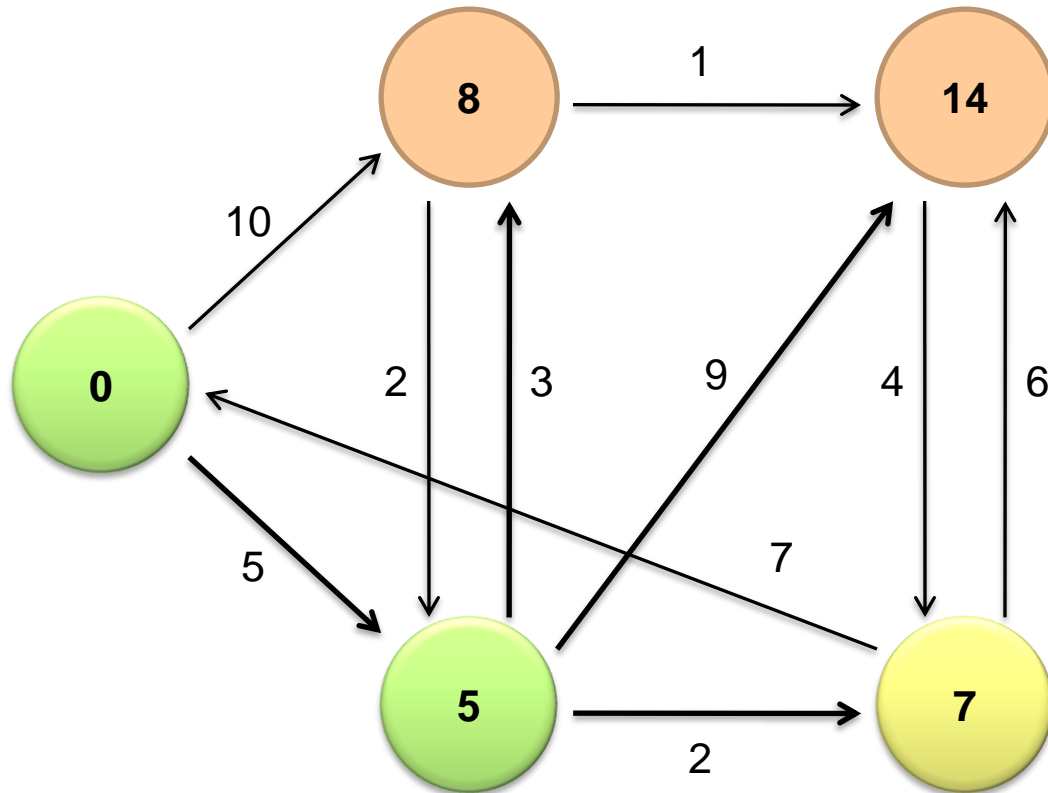
# Dijkstra's Algorithm Example



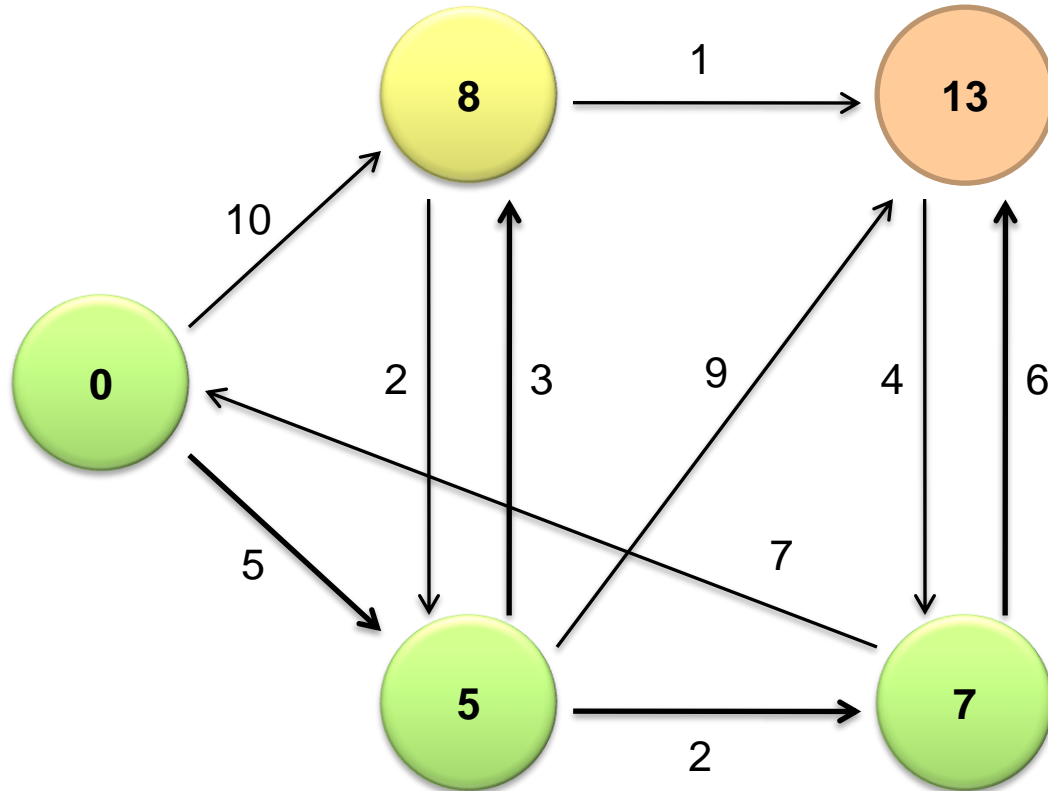
# Dijkstra's Algorithm Example



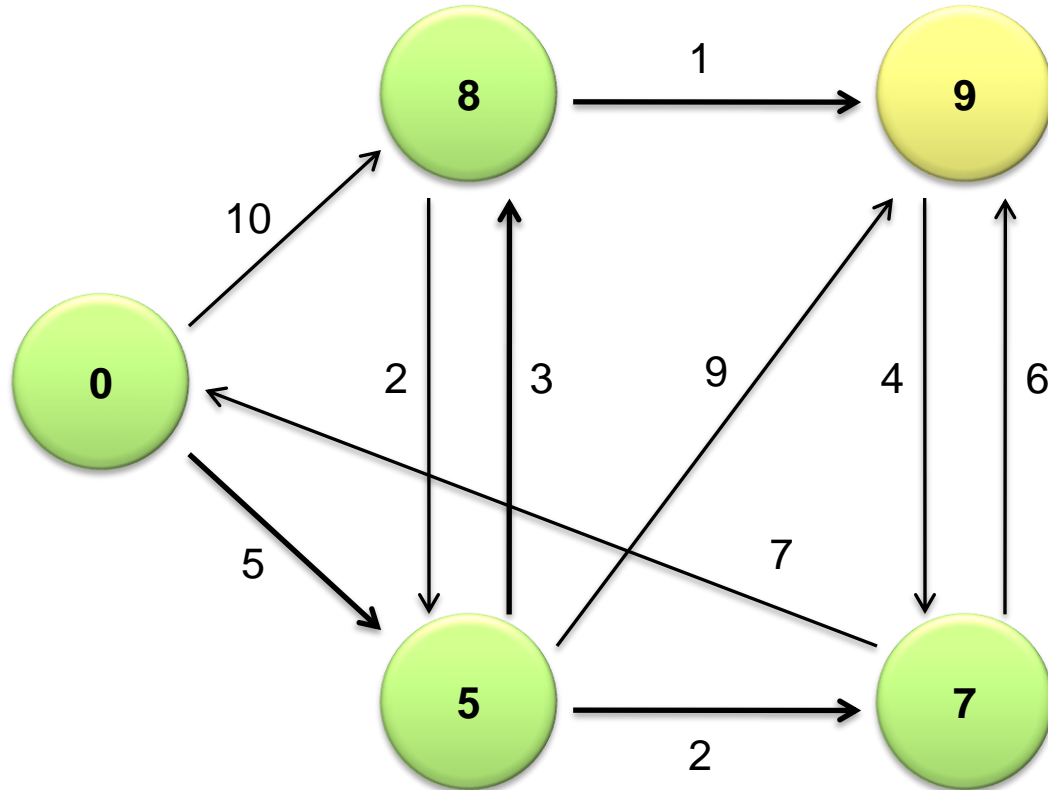
# Dijkstra's Algorithm Example



# Dijkstra's Algorithm Example

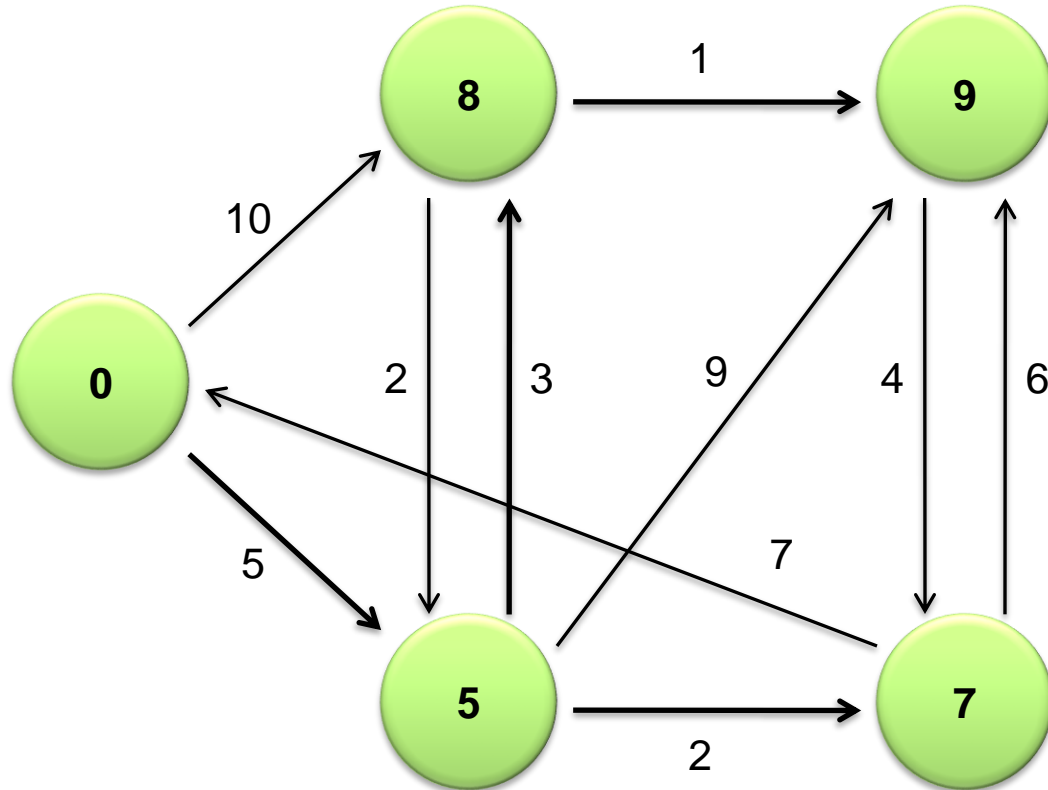


# Dijkstra's Algorithm Example





# Dijkstra's Algorithm Example

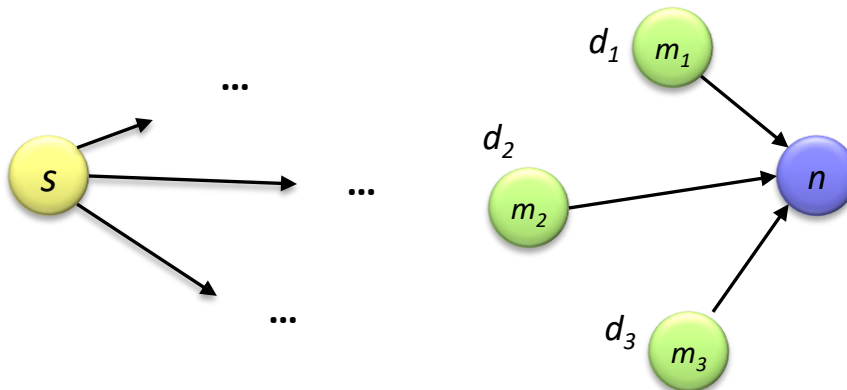


# Single-Source Shortest Path

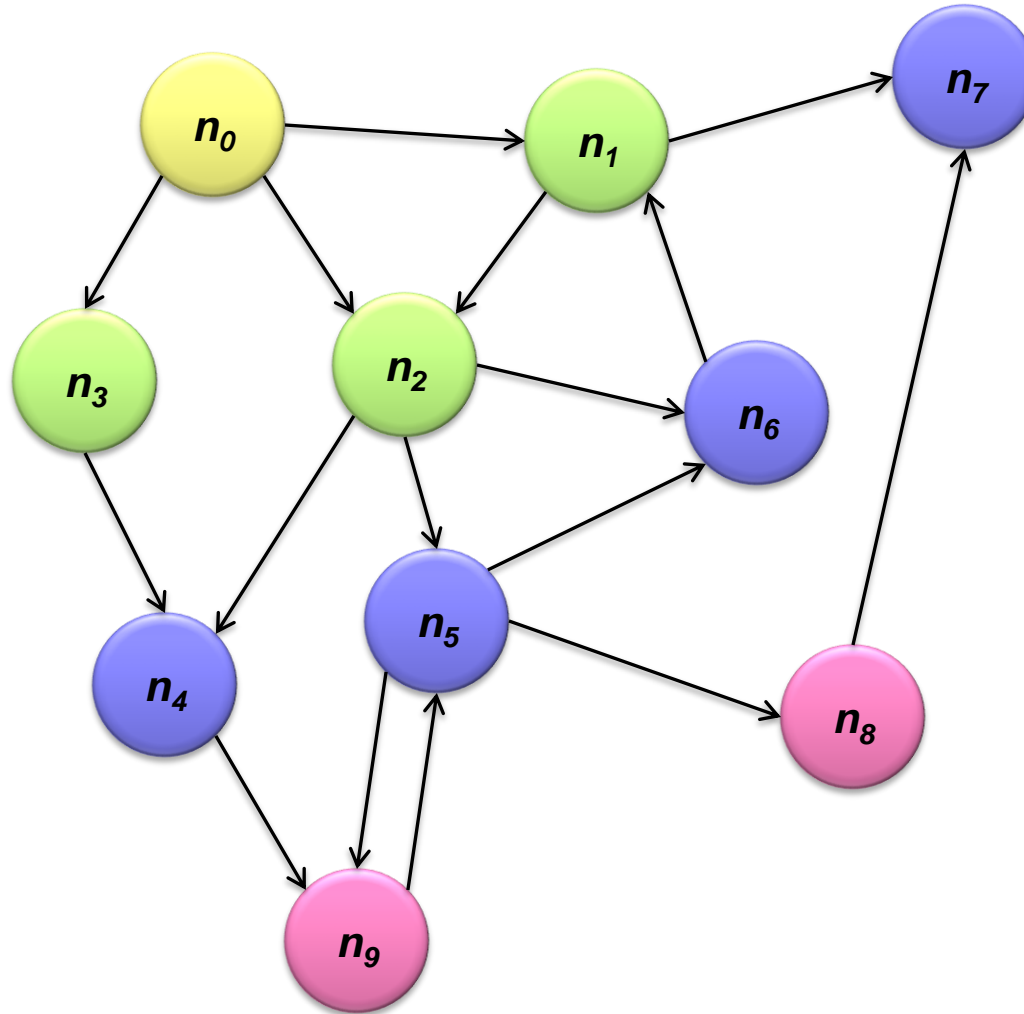
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Single processor machine: Dijkstra's Algorithm
- MapReduce: parallel breadth-first search (BFS)

# Finding the Shortest Path (BFS)

- 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  $a$   
 $\text{DISTANCETo}(s) = 0$
  - For all nodes  $p$  reachable from  $s$ ,  
 $\text{DISTANCETo}(p) = 1$
  - For all nodes  $n$  reachable from some other set of nodes  $M$ ,  
 $\text{DISTANCETo}(n) = 1 + \min(\text{DISTANCETo}(m), m \in M)$



# Visualizing Parallel BFS



# From Intuition to Algorithm

## ○ Data representation:

- Key: node  $n$
- Value:  $d$  (distance from start), adjacency list (nodes reachable from  $n$ )
- Initialization: for all nodes except for start node,  $d = \infty$

## ○ Mapper:

- $\forall m \in \text{adjacency list: emit } (m, d + 1)$
- Remember to also emit distance to yourself

## ○ 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 “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  $(n, \text{Node structure})$  as well
    - where Node = (Adjacency list, distance)

# BFS Pseudo-Code

```
1: class MAPPER
2:   method MAP(nid  $n$ , node  $N$ )
3:      $d \leftarrow N.DISTANCE$ 
4:     EMIT(nid  $n$ ,  $N$ )                                ▷ Pass along graph structure
5:     for all nodeid  $m \in N.ADJACENCYLIST$  do
6:       EMIT(nid  $m$ ,  $d + 1$ )                            ▷ Emit distances to reachable nodes

1: class REDUCER
2:   method REDUCE(nid  $m$ , [ $d_1, d_2, \dots$ ])
3:      $d_{min} \leftarrow \infty$ 
4:      $M \leftarrow \emptyset$ 
5:     for all  $d \in \text{counts } [d_1, d_2, \dots]$  do
6:       if ISNODE( $d$ ) then
7:          $M \leftarrow d$                                 ▷ Recover graph structure
8:       else if  $d < d_{min}$  then                          ▷ Look for shorter distance
9:          $d_{min} \leftarrow d$ 
10:     $M.DISTANCE \leftarrow d_{min}$                         ▷ Update shortest distance
11:    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?
- Practicalities of implementation in MapReduce



# Comparison to Dijkstra

- Dijkstra's algorithm is more efficient
  - At each step, only pursues edges from minimum-cost path inside 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?

# Single Source: Weighted Edges

- Now add positive weights to the edges
  - Why can't edge weights be negative?
- Simple change: add weight  $w$  for each edge in adjacency list
  - In mapper, emit  $(m, d + w_p)$  instead of  $(m, d + 1)$  for each node  $m$

- That's it?

```
1: class MAPPER
2:   method MAP(nid n, node N)
3:      $d \leftarrow N.DISTANCE$ 
4:     EMIT(nid n, N)                                ▷ Pass along graph structure
5:     for all nodeid m ∈ N.ADJACENCYLIST do
6:       EMIT(nid m, d + 1)                          ▷ Emit distances to reachable nodes

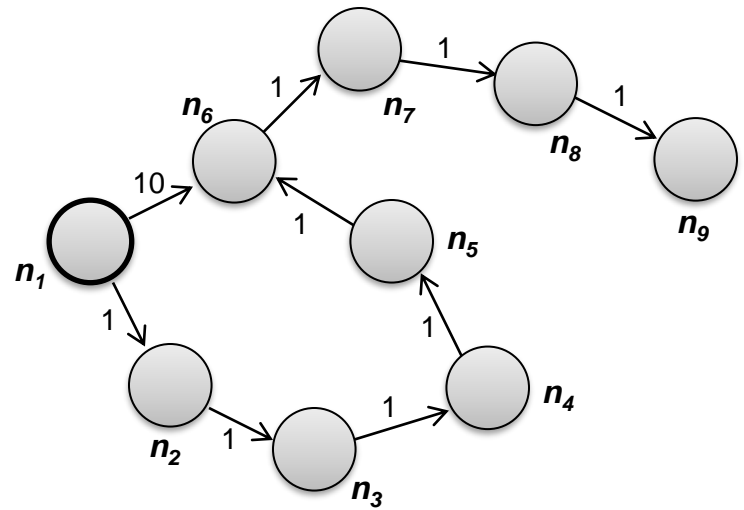
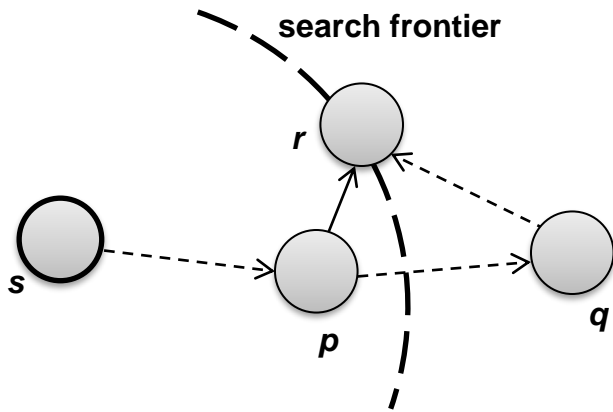
1: class REDUCER
2:   method REDUCE(nid m, [d1, d2, ...])
3:      $d_{min} \leftarrow \infty$ 
4:      $M \leftarrow \emptyset$ 
5:     for all d ∈ counts [d1, d2, ...] do
6:       if ISNODE(d) then
7:          $M \leftarrow d$                                 ▷ Recover graph structure
8:       else if d < dmin then                          ▷ Look for shorter distance
9:          $d_{min} \leftarrow d$ 
10:    M.DISTANCE ← dmin                                ▷ Update shortest distance
11:    EMIT(nid m, node M)
```

# Stopping Criterion

- 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



# Stopping Criterion

- How many iterations are needed in parallel BFS (positive edge weight case)?
- Practicalities of implementation in MapReduce

# Social Search

- When searching, how to rank friends named “John”?
  - Assume undirected graphs
  - Rank matches by distance to user
- Naïve implementations:
  - Precompute all-pairs distances
  - Compute distances at query time
- Can we do better?

# All-Pairs?

- Floyd-Warshall Algorithm: difficult to MapReduce-ify...
- Multiple-source shortest paths in MapReduce: run multiple parallel BFS *simultaneously*
  - Assume source nodes  $\{s_0, s_1, \dots, s_n\}$
  - Instead of emitting a single distance, emit an array of distances, with respect to each source
  - Reducer selects minimum for each element in array
- Does this scale?
  - What is the problem?

# Landmark Approach (aka sketches)

- Select  $k$  seeds  $\{s_0, s_1, \dots s_k\}$
- Compute distances from seeds to every node:
  - A = [2, 1, 1]
  - B = [1, 1, 2]
  - C = [4, 3, 1]
  - D = [1, 2, 4]
- What can we conclude about distances?
  - Insight: landmarks bound the maximum path length
- Lots of details:
  - How to more tightly bound distances
  - How to select landmarks
- Use multi-source parallel BFS implementation in MapReduce!



# Graphs and MapReduce

- A large class of graph algorithms 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

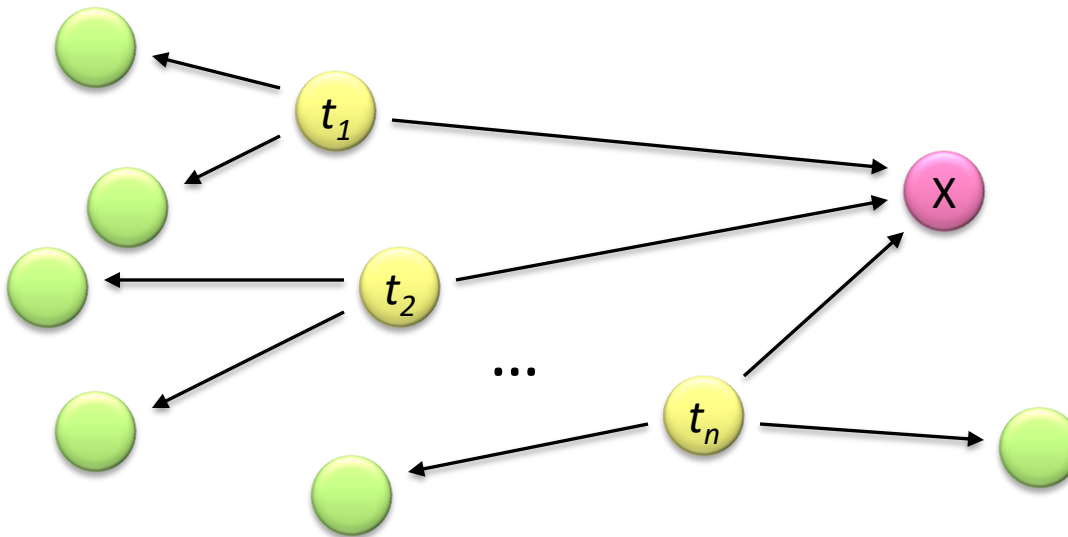
- Random surfer model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- PageRank
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages
- PageRank captures notions of page importance
  -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

# PageRank: Defined

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

- $C(t)$  is the out-degree of  $t$
- $\alpha$  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

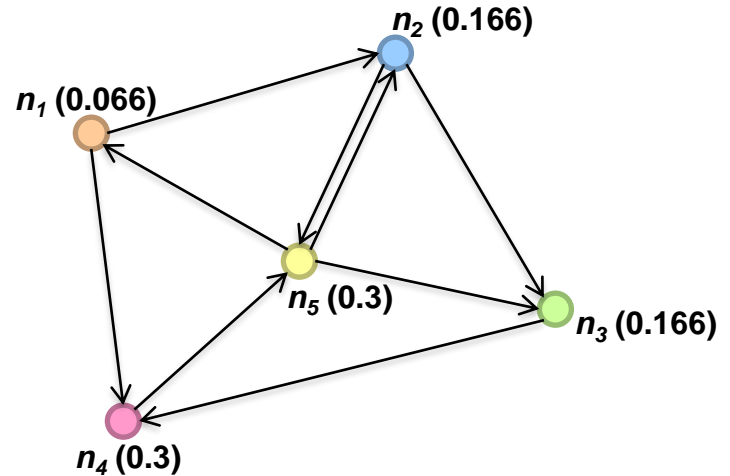
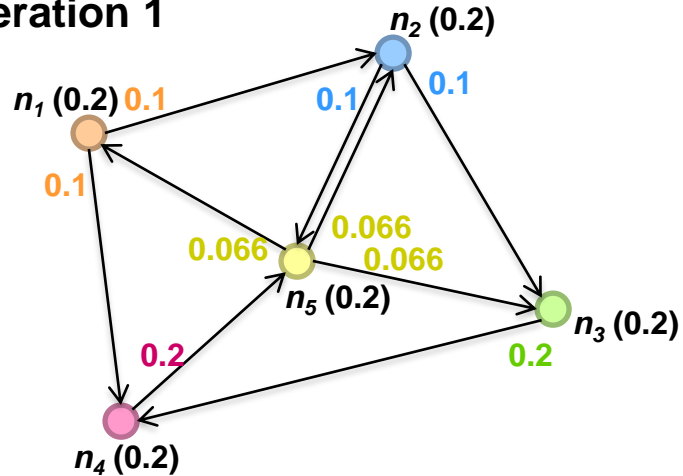
- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm:
  - Start with seed  $PR_i$  values
  - Each page distributes  $PR_i$  “credit” to all pages it links to
  - Each target page adds up “credit” from multiple in-bound links to compute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

# Simplified PageRank

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

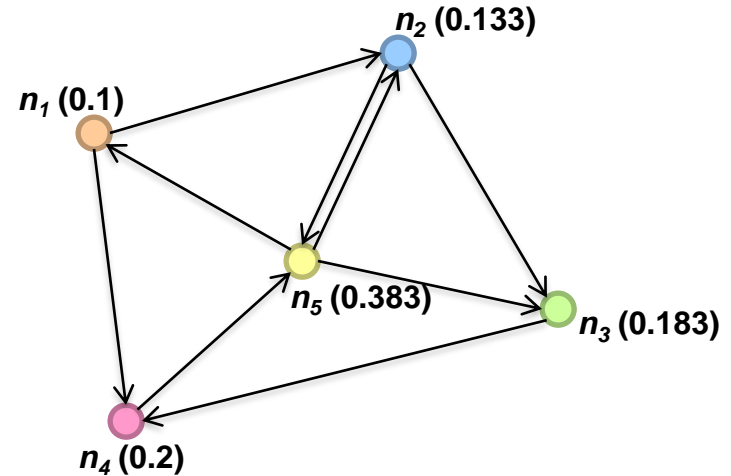
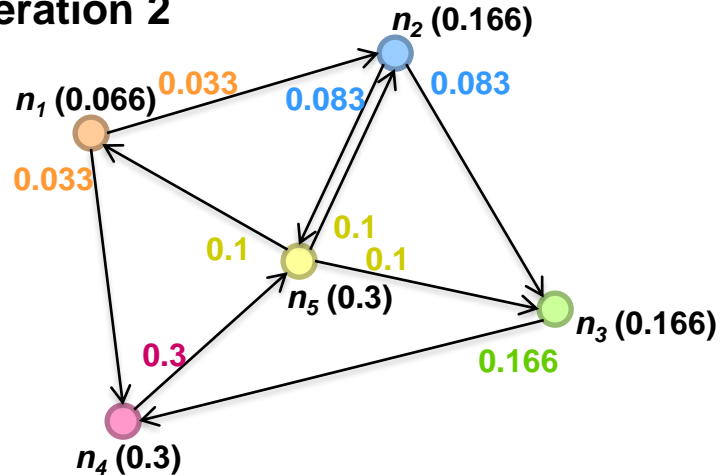
# Sample PageRank Iteration (1)

Iteration 1

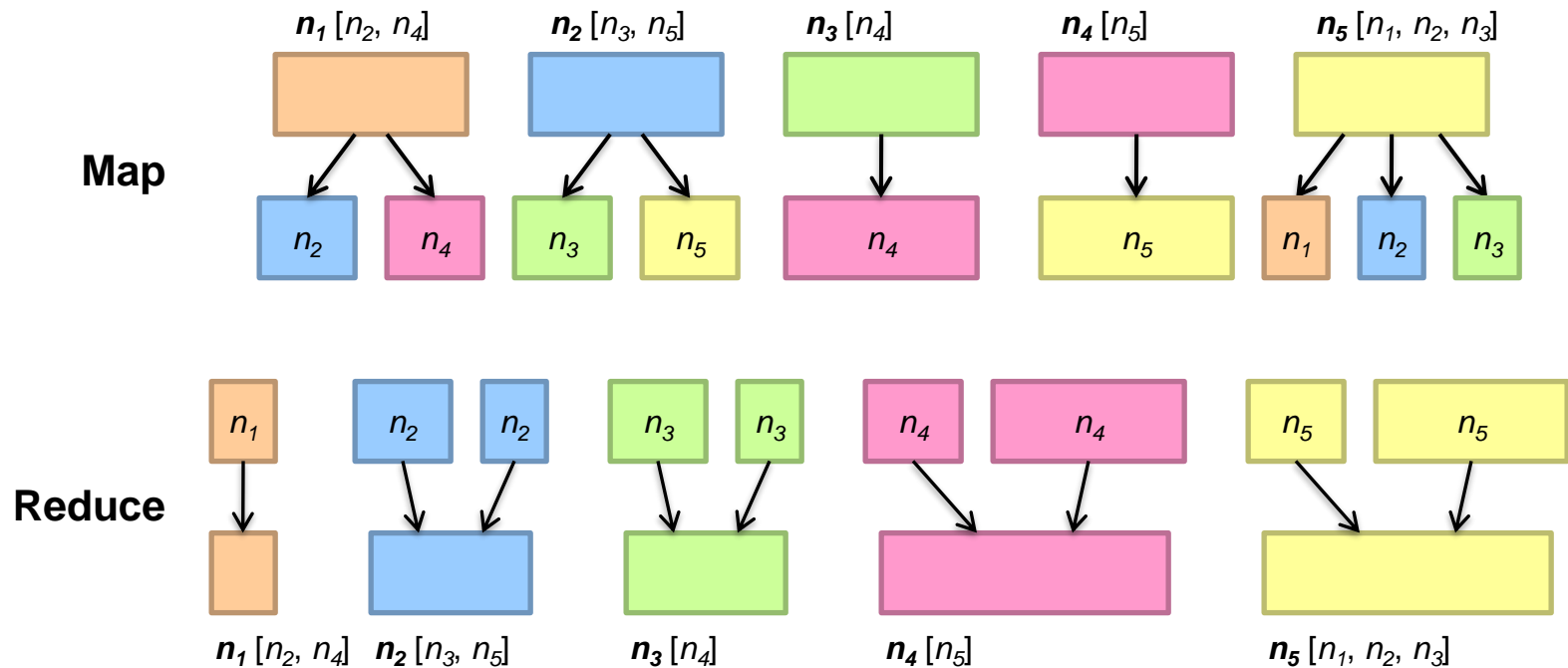


# Sample PageRank Iteration (2)

Iteration 2



# PageRank in MapReduce





# PageRank Pseudo-Code

```
1: class MAPPER
2:   method MAP(nid  $n$ , node  $N$ )
3:      $p \leftarrow N.\text{PAGERANK} / |N.\text{ADJACENCYLIST}|$ 
4:      $\text{EMIT}(\text{nid } n, N)$  ▷ Pass along graph structure
5:     for all nodeid  $m \in N.\text{ADJACENCYLIST}$  do
6:        $\text{EMIT}(\text{nid } m, p)$  ▷ Pass PageRank mass to neighbors
7:
8: class REDUCER
9:   method REDUCE(nid  $m$ , [ $p_1, p_2, \dots$ ])
10:     $M \leftarrow \emptyset$ 
11:    for all  $p \in \text{counts } [p_1, p_2, \dots]$  do
12:      if  $\text{ISNODE}(p)$  then
13:         $M \leftarrow p$  ▷ Recover graph structure
14:      else
15:         $s \leftarrow s + p$  ▷ Sums incoming PageRank contributions
16:     $M.\text{PAGERANK} \leftarrow s$ 
17:     $\text{EMIT}(\text{nid } m, \text{node } M)$ 
```

# PageRank vs. BFS

	PageRank	BFS
Map	$PR/N$	$d+1$
Reduce	sum	min

# 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

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

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

- Additional optimization: make it a single pass!

# 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?
  - Not a straightforward question
- Watch out for link spam:
  - Link farms
  - Spider traps
  - ...

# Beyond PageRank

- Variations of PageRank
  - Weighted edges
  - Personalized PageRank
- Variants on graph random walks
  - Hubs and authorities (HITS)
  - SALSA

# Applications

- Static prior for web ranking
- Identification of “special nodes” in a network
- Link recommendation
- Additional feature in any machine learning problem

# Other Classes of Graph Algorithms

- Subgraph pattern matching
- Computing simple graph statistics
  - Degree vertex distributions
- Computing more complex graph statistics
  - Clustering coefficients
  - Counting triangles (soon....)

# General Issues for Graph Algorithms

- Sparse vs. dense graphs
- Graph topologies



# MapReduce Sucks

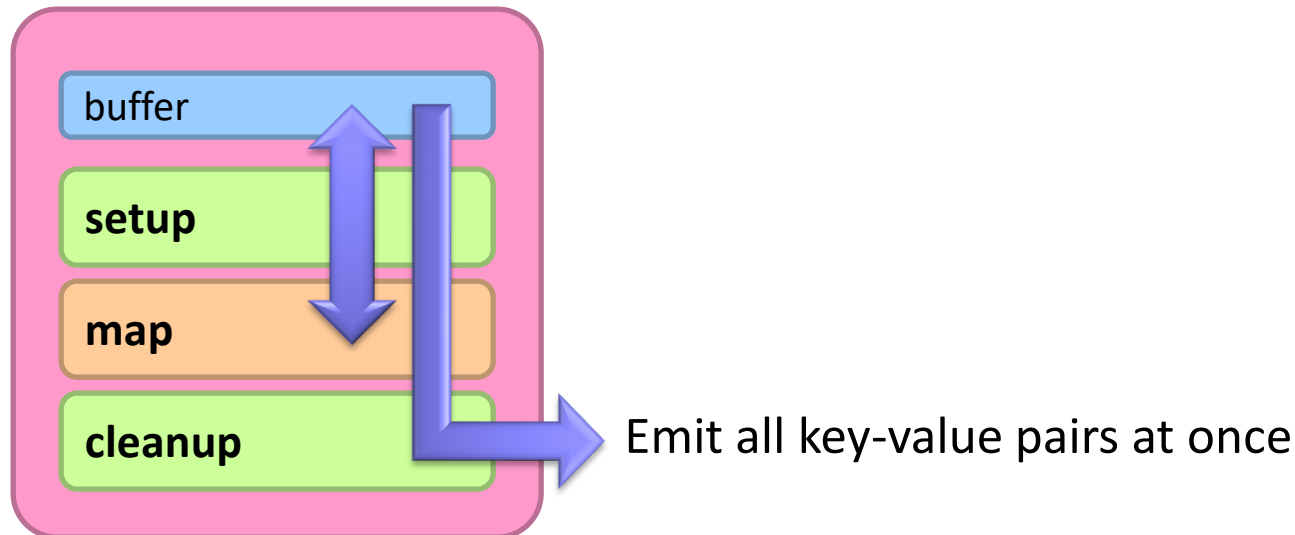
- Java verbosity
- Hadoop task startup time
- Stragglers
- Needless graph shuffling
- Checkpointing at each iteration

# MapReduce sucks at iterative algorithms

- Alternative programming models (later)
- Easy fixes (now)

# In-Mapper Combining

- Use combiners
  - Perform local aggregation on map output
  - Downside: intermediate data is still materialized
- Better: in-mapper combining
  - Preserve state across multiple map calls, aggregate messages in buffer, emit buffer contents at end
  - Downside: requires memory management



# 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

# Today's Agenda

- Graph problems and representations
- Parallel breadth-first search
- PageRank
- Beyond PageRank and other graph algorithms
- Optimizing graph algorithms