



# Graph Algorithms with MapReduce

S4230

Jay Urbain, Ph.D.

Credits:

- MapReduce: The Definitive Guide, Tom White
- Jeffery Dean and Sanjay Chemawat. *MapRecuce*
- Jimmy Lin and Chris Dyer. *Data Intensive Text Processing with MapReduce*

# Today' s Topics

- Introduction to graph algorithms and graph representations
- Single Source Shortest Path (SSSP) problem
  - Refresher: Dijkstra' s algorithm
  - Breadth-First Search with MapReduce
- PageRank

# 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

# Some Graph Problems

- Finding shortest paths
  - Routing Internet traffic and UPS trucks
- Finding minimum spanning trees
  - Telco laying down optical 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
- PageRank, HITS, EdgeRank

# Graphs and MapReduce

- Graph algorithms typically involve:
  - Performing computation at each node
  - Processing node-specific data, edge-specific data, and link structure
  - Traversing the graph in some manner
- Key questions:
  - How do you represent graph data in MapReduce?
  - How do you traverse a graph in MapReduce?

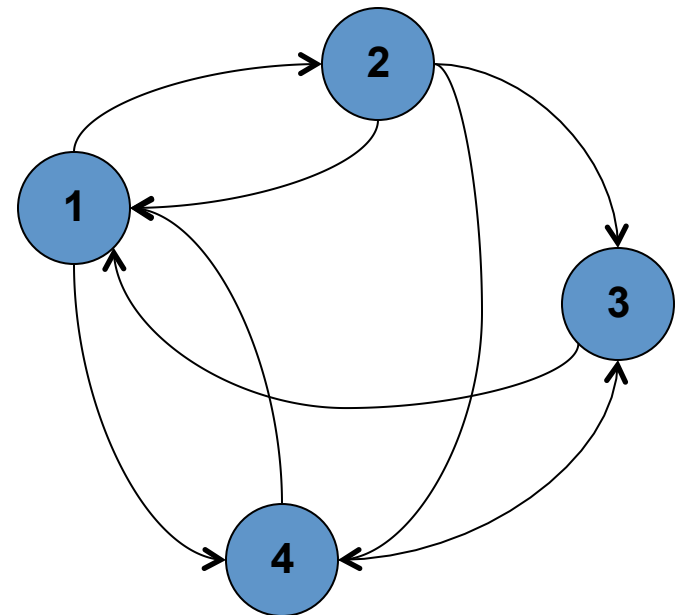
# Representation Graphs

- $G = (V, E)$ 
  - A poor representation for computational purposes
- 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	0	1	0	0
4	1	0	1	0



# Adjacency Matrices: Critique

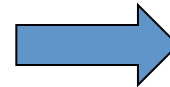
- Advantages:
  - Naturally encapsulates iteration over nodes
  - Rows and columns correspond to *in-links* and *out-links*
  - Simple, fast access
- Disadvantages:
  - Lots of zeros for sparse matrices
  - Lots of wasted space



# Adjacency Lists

- Take adjacency matrices... and throw away all the zeros
- Represent only *out-links* from a node – “*sparse matrix representation*”

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



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

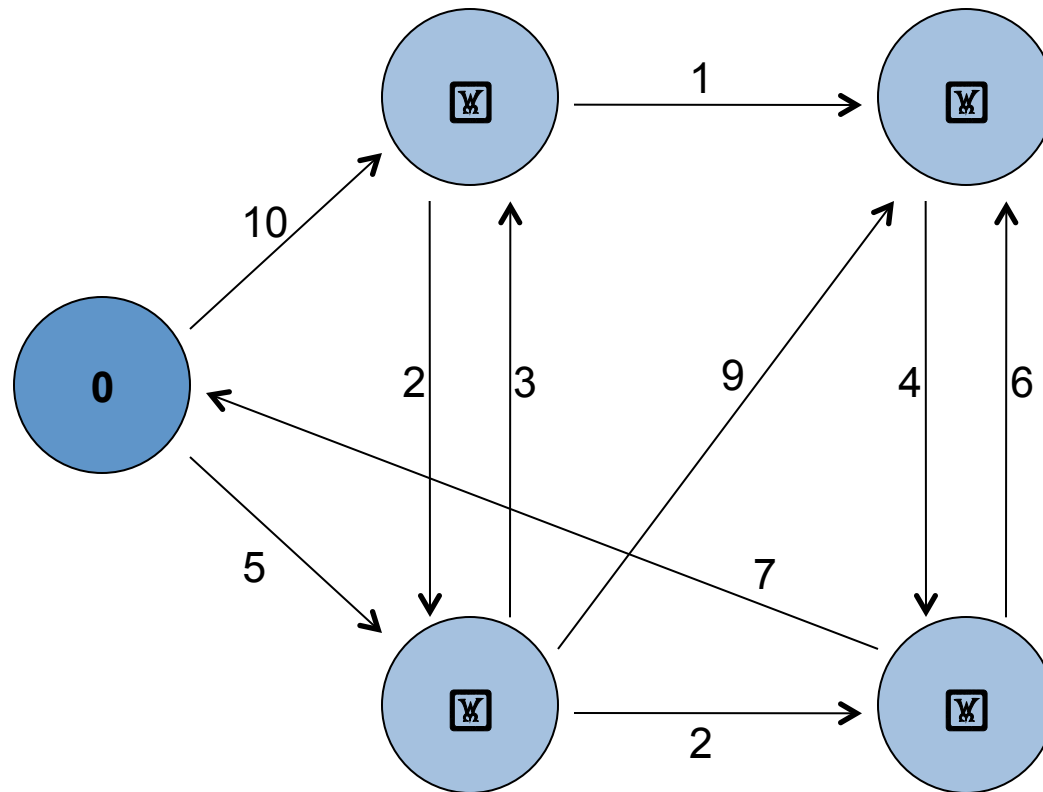
# Adjacency Lists: Critique

- Advantages:
  - Much more compact representation
  - Easy to compute over *out-links*
  - Graph structure can be broken up and distributed
- Disadvantages:
  - More difficult to compute over *in-links*

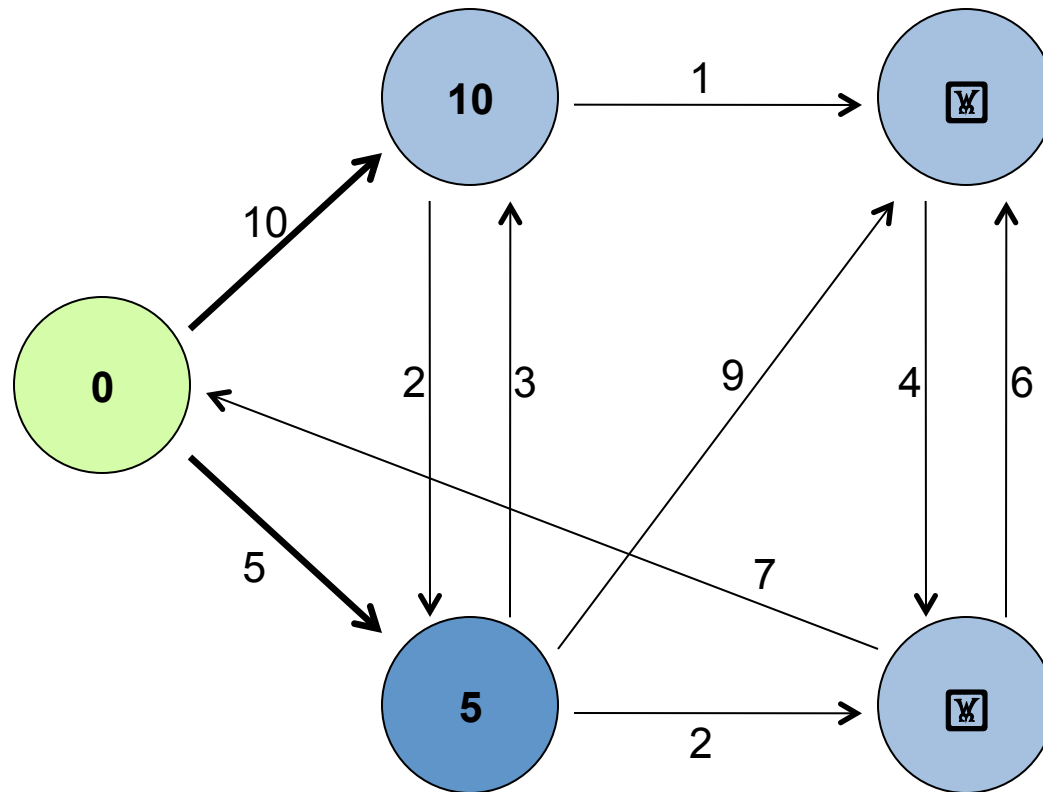
# Single Source Shortest Path

- **Problem:** find shortest path from a source node to one or more target nodes
- First, a refresher: Dijkstra's Algorithm

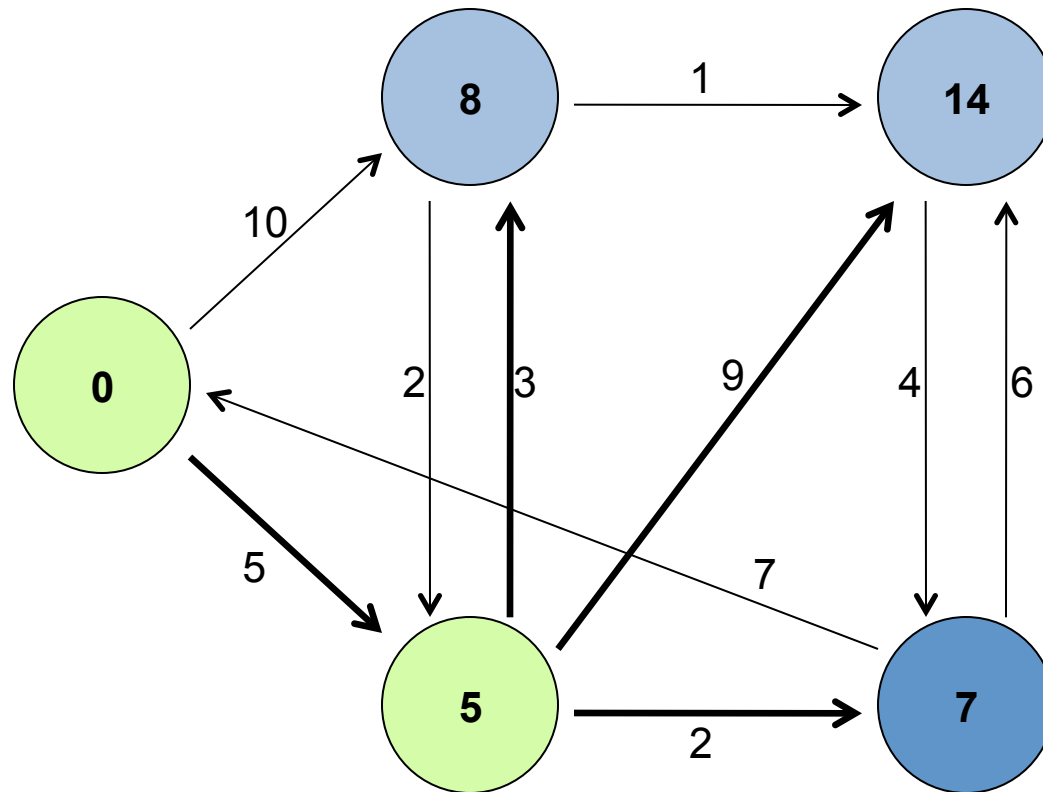
# Dijkstra's Algorithm Example



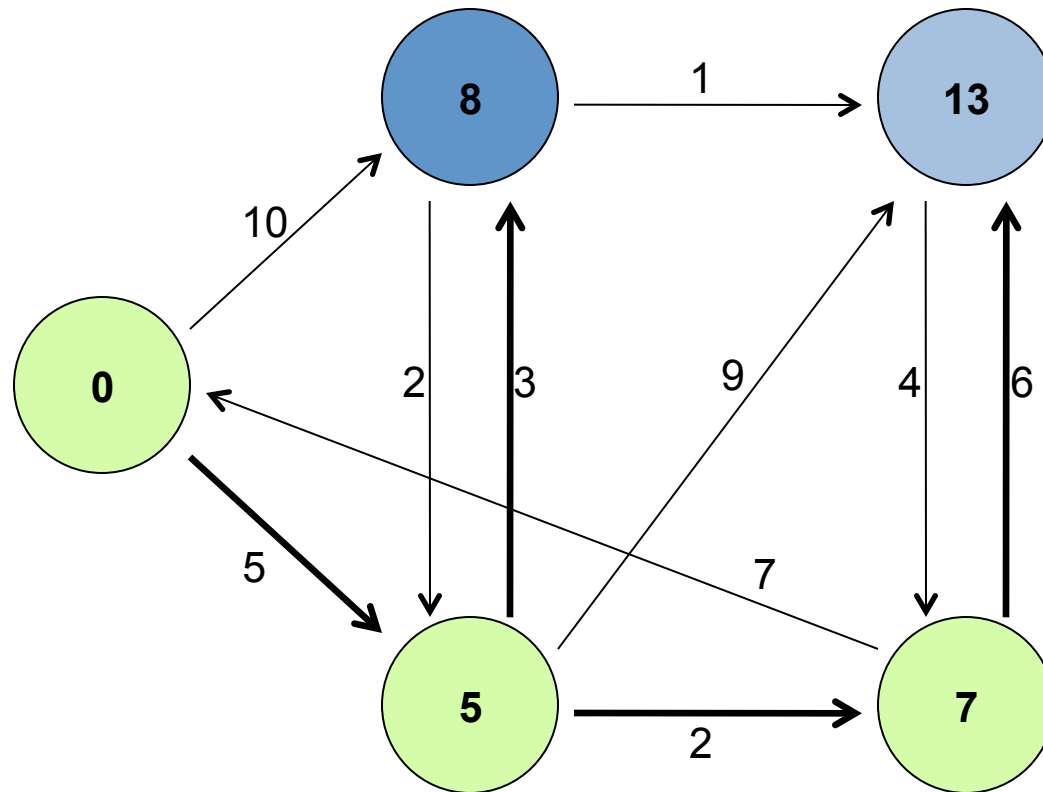
# Dijkstra's Algorithm Example



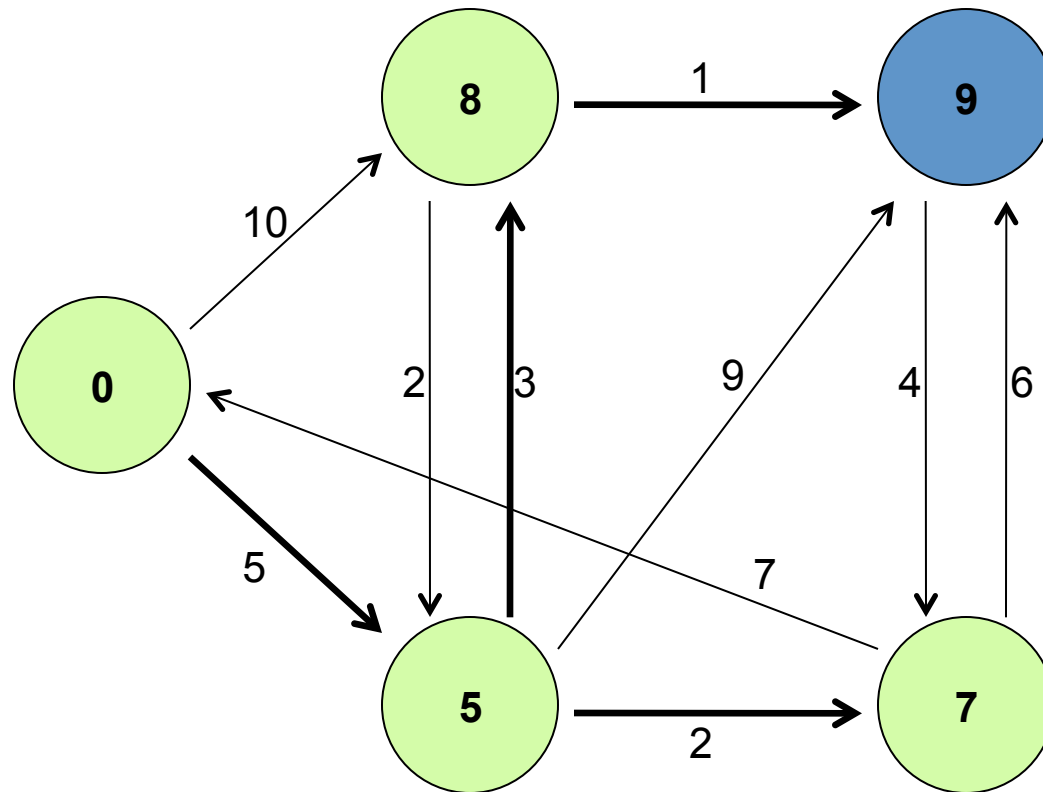
# Dijkstra's Algorithm Example



# Dijkstra's Algorithm Example

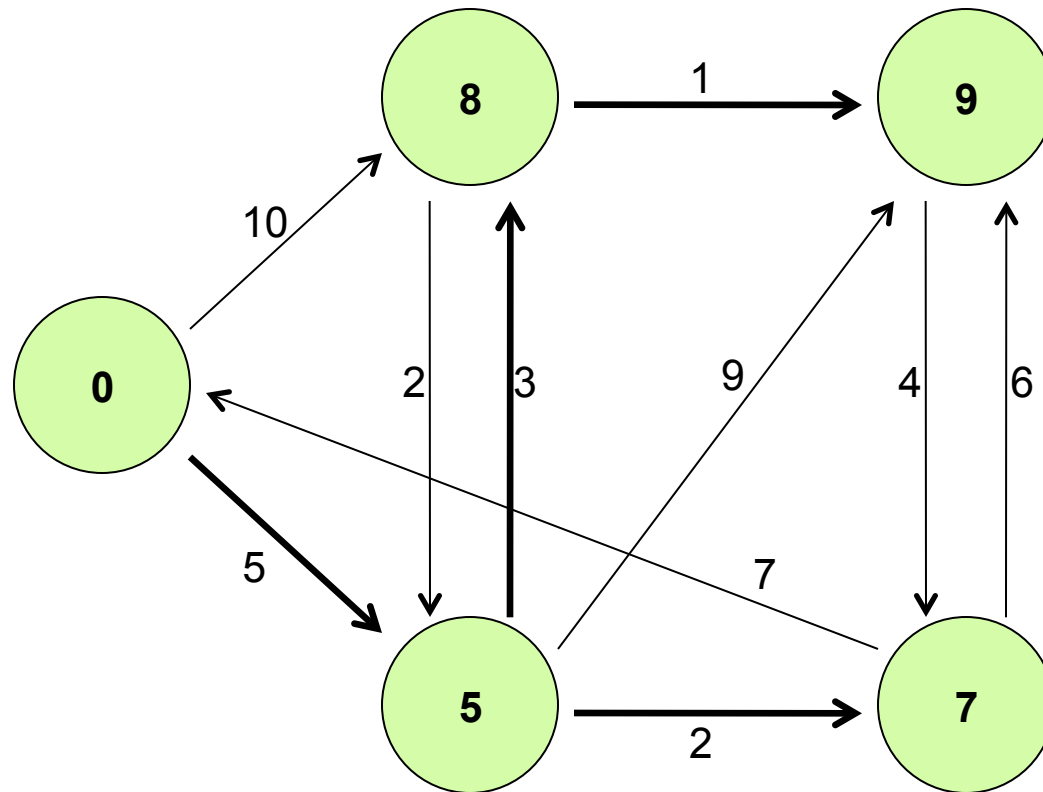


# Dijkstra's Algorithm Example





# Dijkstra's Algorithm Example



# Single Source Shortest Path

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

# Finding the Shortest Path

- First, consider equal edge weights
- Solution to the problem can be defined inductively
- Here's the intuition:
  - $\text{DistanceTo}(\text{startNode}) = 0$
  - For all nodes  $n$  directly reachable from  $\text{startNode}$ ,  $\text{DistanceTo}(n) = 1$
  - For all nodes  $n$  reachable from  $m$  or some other set of nodes  $S$ ,  $\text{DistanceTo}(n) = 1 + \min(\text{DistanceTo}(m), \text{DistanceTo}(s) \mid m \text{ or } s \text{ is an element of } S)$

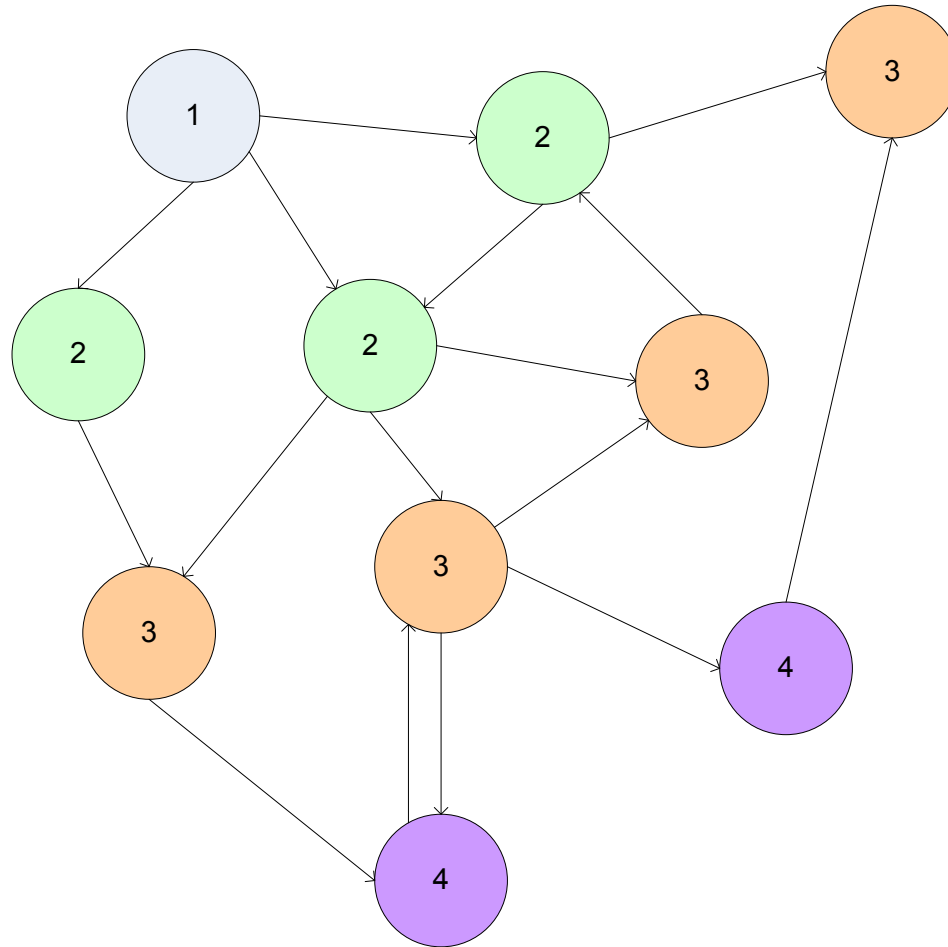
# From Intuition to Algorithm

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

# Multiple Iterations Needed

- This MapReduce 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
- Preserving graph structure:
  - Problem: Where did the *points-to* list go?
  - **Solution: Mapper emits ( $n$ , points-to) as well**

# Visualizing Parallel BFS



# Termination

- Does the algorithm ever terminate?
  - Eventually, all nodes will be discovered, all edges will be considered (in a connected graph)
- When do we stop?

# 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  $p$
- Does this ever terminate?
  - Yes! Eventually, no better distances will be found. When distance is the same, we stop
  - Mapper should emit  $(n, D)$  to ensure that “current distance” is carried into the reducer



## Graph

- a: b, c
- b: c, d
- c:
- d:
- e:

Mapper ( a, (0, (b,c)))

Emit( b, (1, (c,d)))

Emit( c, (1, ()))

...

Reducer ( b, (1, (c,d)))

(b,1)<-min(b,1)

output(b, (1, (c,d)))

Reducer ( c, (1, ()))

(c,1)<- min(c,1)

output(c, (1, ()))

Mapper ( b, (1, (c,d)))

Emit( c, (2, ()))

Emit( d, (2, ()))

...

Reducer ( c, (2, ()))

(c,1)<- min(c,2)

// no output

Reducer ( d, (2, ()))

// no output

(d,1)<- min(d,2)

// no output

# 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
  - Divide and conquer
  - **Throw more hardware at the problem!**

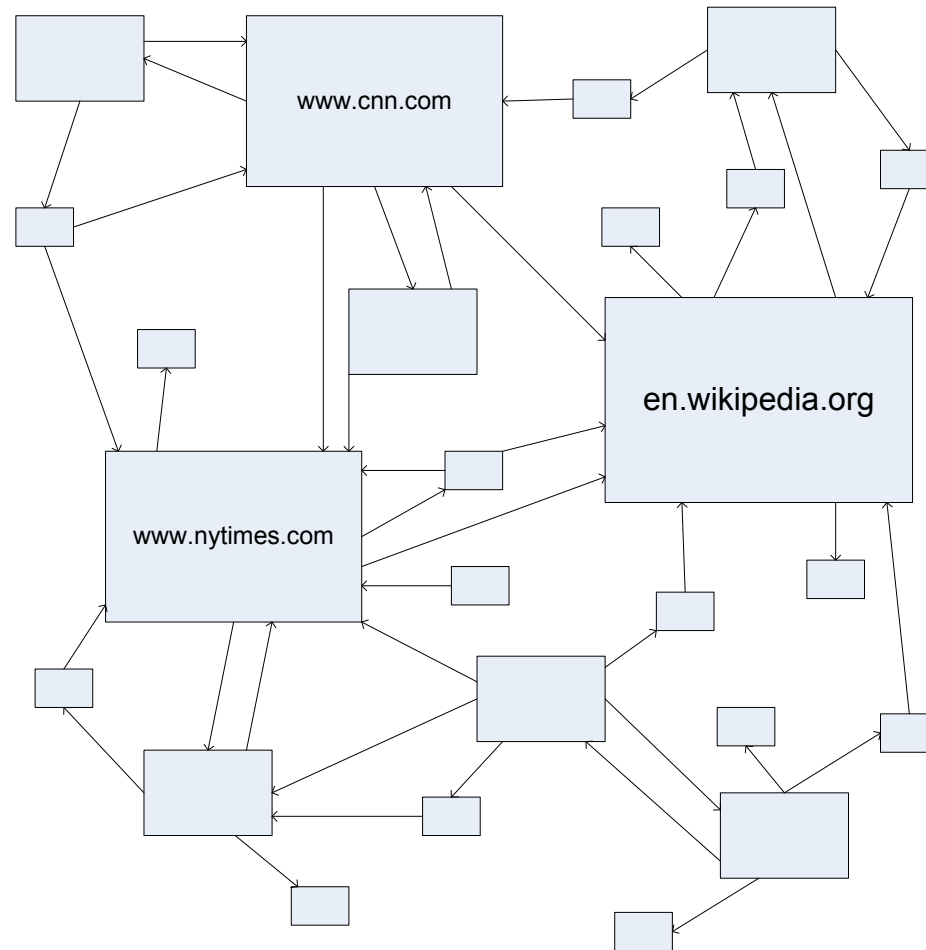
# General Approach

- MapReduce is adept at manipulating graphs
  - Store graphs as adjacency lists
- 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:
  - **Remember to “pass” graph structure from one iteration to next**

# Random Walks Over the Web

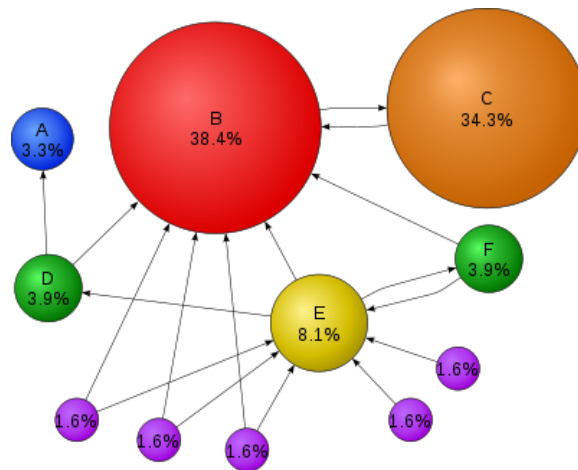
- Model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- What's the amount of time that will be spent on any given page?
- This is PageRank

# PageRank: Visually



# PageRank

- Initially developed at Stanford University by Google founders, Larry Page and Sergey Brin, in 1995.
- Program implemented by Google to rank any type of recursive “documents” using MapReduce.
- Led to a functional prototype named Google in 1998.
- Still provides an important function for Google's web search tools.



# PageRank

- Assume a small universe of four web pages: **A**, **B**, **C** and **D**. The initial approximation of PageRank would be evenly divided between these four documents.
- Each document would begin with an estimated PageRank of 0.25.
- If the only links in the system were from pages **B**, **C**, and **D** to **A**, each link would transfer 0.25 PageRank to **A** upon the next iteration, for a total of 0.75.

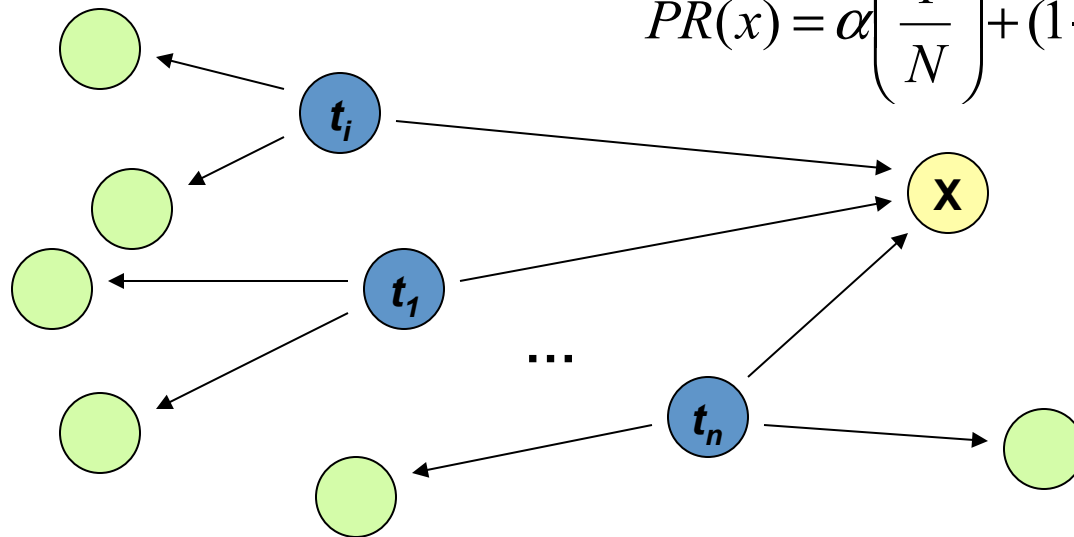
$$PR(A) = PR(B) + PR(C) + PR(D).$$

# PageRank: Defined

- Given page  $x$  with in-bound links  $t_1...t_n$ , 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

- We can define PageRank as:

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





# PageRank

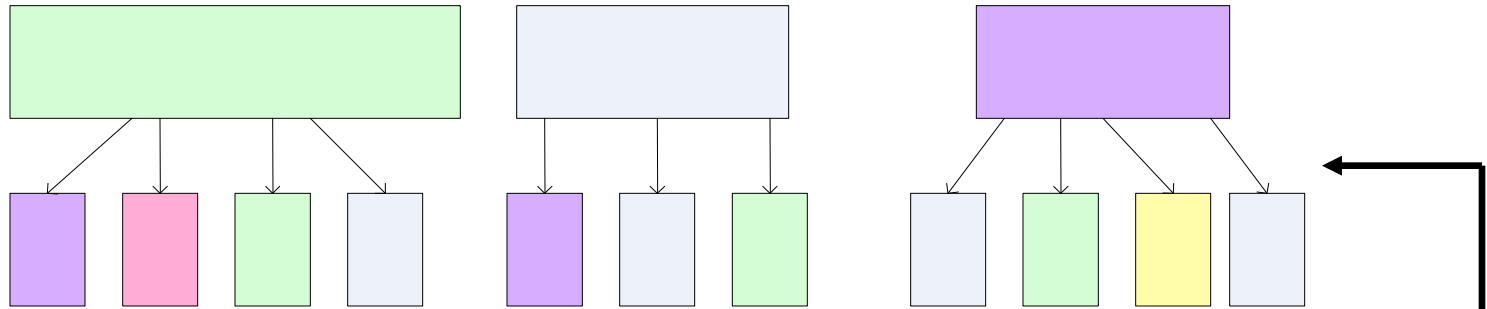
- Simulates a “random-surfer”
- Begins with pair (URL, list-of-URLs)
- Maps to (URL, (PR, list-of-URLs))
- Maps again taking above data, and for each  $u$  in *list-of-URLs* returns  $(u, PR/|list-of-URLs|)$ , as well as  $(u, new-list-of-URLs)$
- Reduce receives (URL, list-of-URLs), and many (URL, value) pairs and calculates (URL, (new-PR, list-of-URLs))

# Computing PageRank

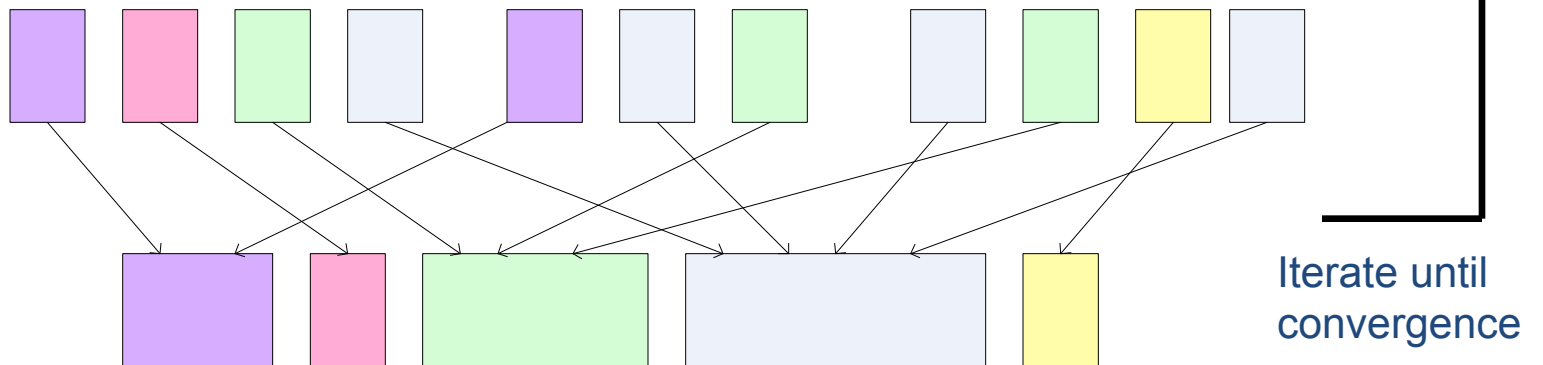
- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration is local
- Sketch of algorithm:
  - Start with seed  $PR_i$  values
  - Each page distributes  $PR_i$  “credit” to all pages it links to
  - Each target page adds up “credit” from multiple in-bound links to compute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

# PageRank in MapReduce

**Map:** distribute PageRank “credit” to link targets



**Reduce:** gather up PageRank “credit” from multiple sources to compute new PageRank value



# PageRank: Issues

- Is PageRank guaranteed to converge? How quickly?
- What is the “correct” value of *alpha*, and how sensitive is the algorithm to it?
- What about dangling links?
- How do you know when to stop?