

INGI2145: CLOUD COMPUTING (Fall 2014)

Algorithms in MapReduce

23 October 2014


What we have seen so far

- We saw how the MapReduce model could be used to **filter**, **collect**, and **aggregate** values
- This is useful for data with limited structure
 - We could extract pieces of input data items and collect them to run various reduce operations
 - We could “join” two different data sets on a common key
- But that’s not enough...

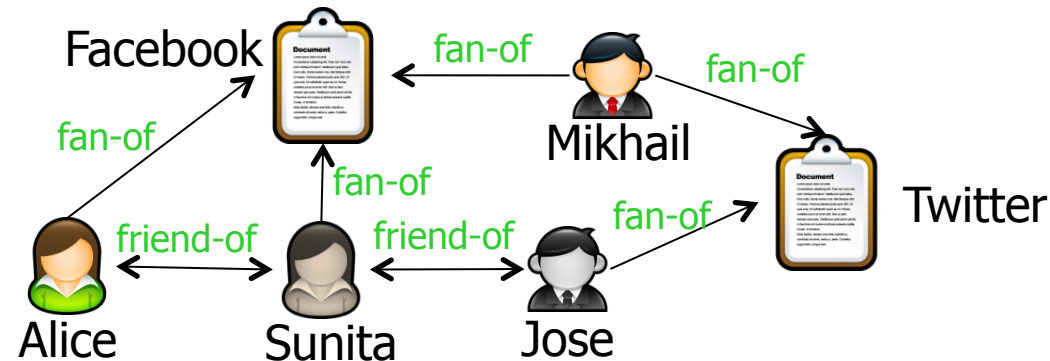
Beyond average/sum/count

- Much of the world is a network of relationships and shared features
 - Members of a social network can be friends, and may have shared interests / memberships / etc.
 - Customers might view similar movies, and might even be clustered by interest groups
 - The Web consists of documents with links
 - Documents are also related by topics, words, authors, etc.
- We need a toolbox of algorithms for analyzing data that has both relationships and properties

Plan for today

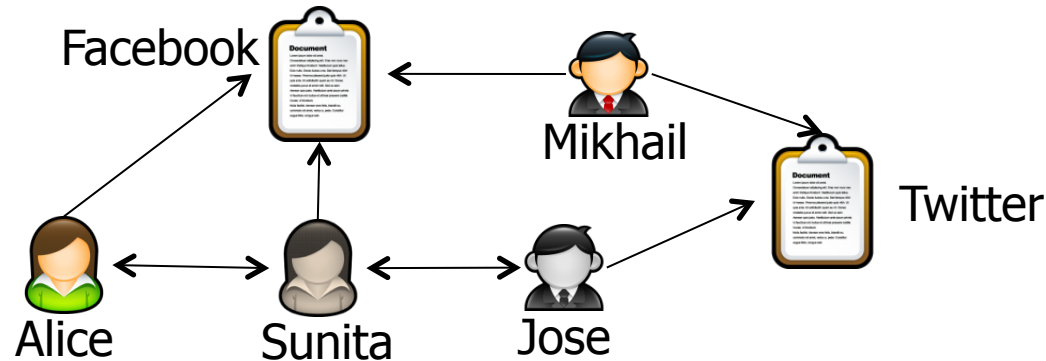
- Representing data in graphs 
- Graph algorithms in MapReduce
 - Computation model
 - Iterative MapReduce
- A toolbox of algorithms
 - Single-source shortest path (SSSP)
 - k-means clustering
 - Classification with Naïve Bayes
 - PageRank

Thinking about related objects



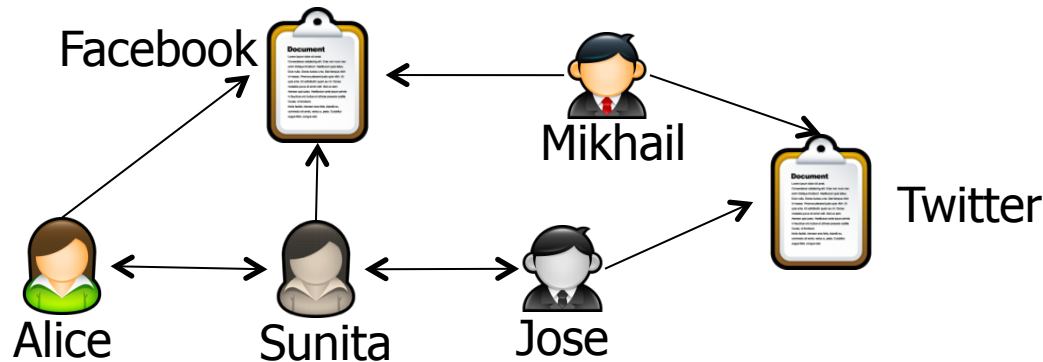
- We can represent related objects as a **labeled, directed graph**
- Entities are typically represented as nodes; relationships are typically edges
 - Nodes all have IDs, and possibly other properties
 - Edges typically have values, possibly IDs and other properties

Encoding the data in a graph



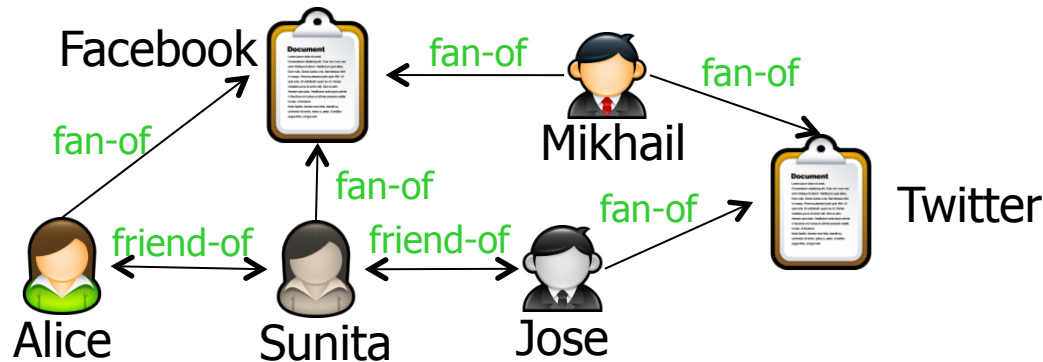
- Recall basic definition of a graph:
 - $G = (V, E)$ where V denotes vertices, E denotes edges of the form (v_1, v_2) where $v_1, v_2 \in V$
- Assume we only care about connected vertices
 - Then we can capture a graph simply as a set of **edges**
 - ... or as an **adjacency list**: v_i goes to $[v_j, v_{j+1}, \dots]$

Graph encodings: Set of edges



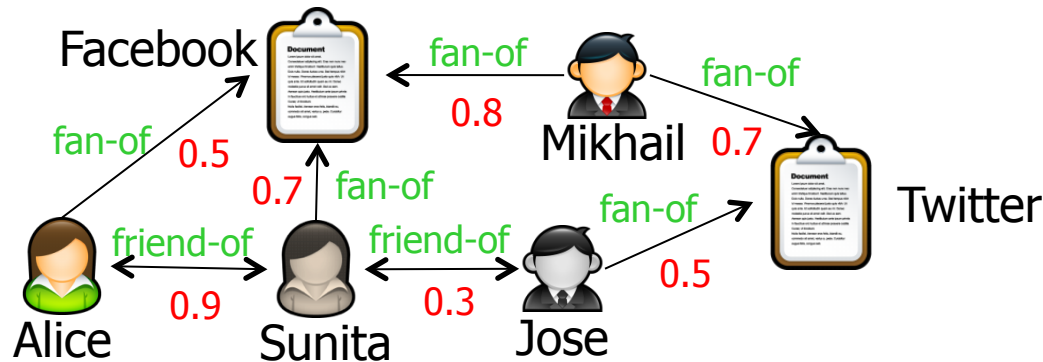
(Alice, Facebook)
(Alice, Sunita)
(Jose, Twitter)
(Jose, Sunita)
(Mikhail, Facebook)
(Mikhail, Twitter)
(Sunita, Facebook)
(Sunita, Alice)
(Sunita, Jose)

Graph encodings: Adding edge types




(Alice, fan-of, Facebook)
(Alice, friend-of, Sunita)
(Jose, fan-of, Twitter)
(Jose, friend-of, Sunita)
(Mikhail, fan-of, Facebook)
(Mikhail, fan-of, Twitter)
(Sunita, fan-of, Facebook)
(Sunita, friend-of, Alice)
(Sunita, friend-of, Jose)

Graph encodings: Adding weights

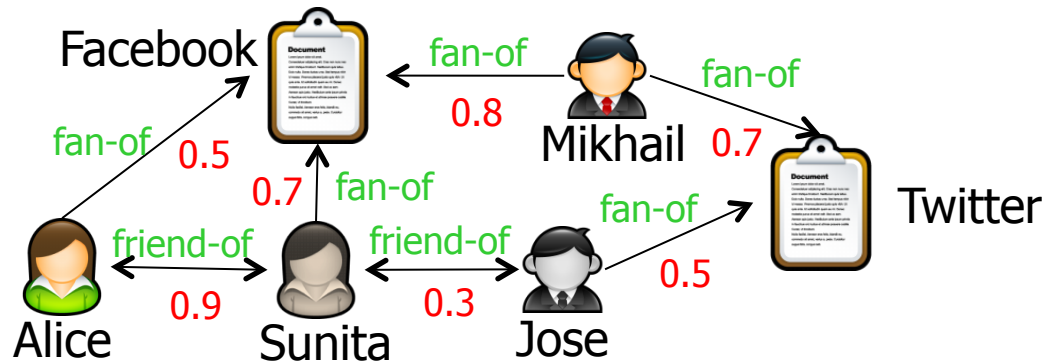


(Alice, fan-of, 0.5, Facebook)
(Alice, friend-of, 0.9, Sunita)
(Jose, fan-of, 0.5, Twitter)
(Jose, friend-of, 0.3, Sunita)
(Mikhail, fan-of, 0.8, Facebook)
(Mikhail, fan-of, 0.7, Twitter)
(Sunita, fan-of, 0.7, Facebook)
(Sunita, friend-of, 0.9, Alice)
(Sunita, friend-of, 0.3, Jose)

Plan for today

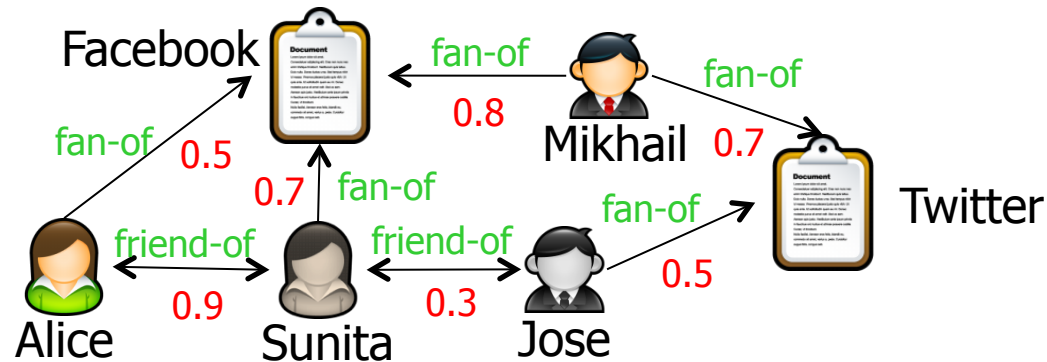
- Representing data in graphs ✓
- Graph algorithms in MapReduce 
 - Computation model
 - Iterative MapReduce
- A toolbox of algorithms
 - Single-source shortest path (SSSP)
 - k-means clustering
 - Classification with Naïve Bayes
 - PageRank

A computation model for graphs



- Once the data is encoded in this way, we can perform various computations on it
 - Simple example: Which users are their friends' best friend?
- This is often done by
 - annotating the vertices with additional information, and
 - propagating the information along the edges
 - “Think like a vertex”!

A computation model for graphs

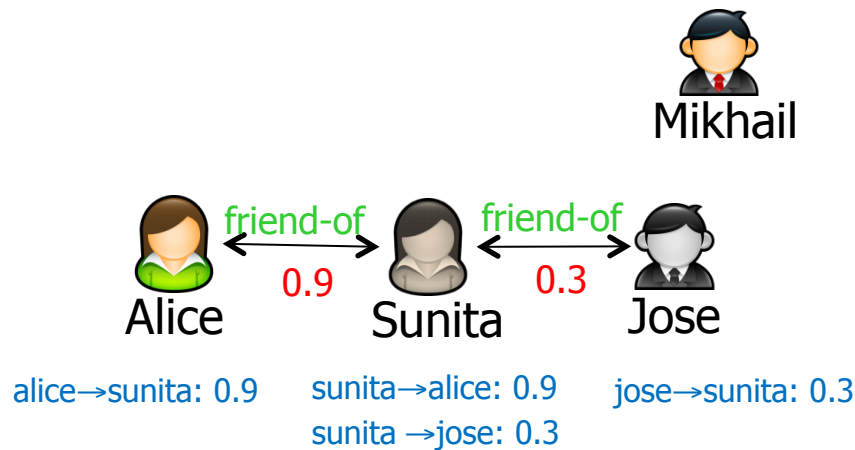


All my friends have me as their best friend
(highest strength edge)

- Example: Am I my friends' best friend?
 - Step #1: Discard irrelevant vertices and edges

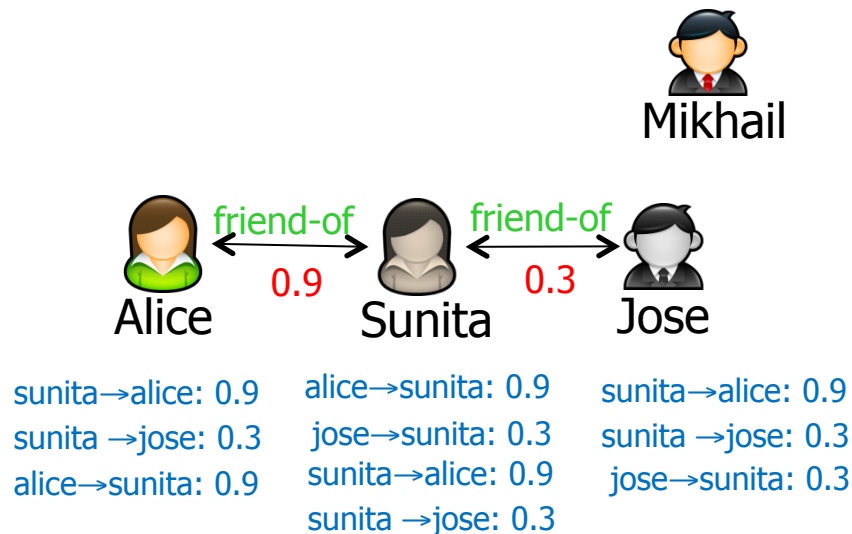


A computation model for graphs



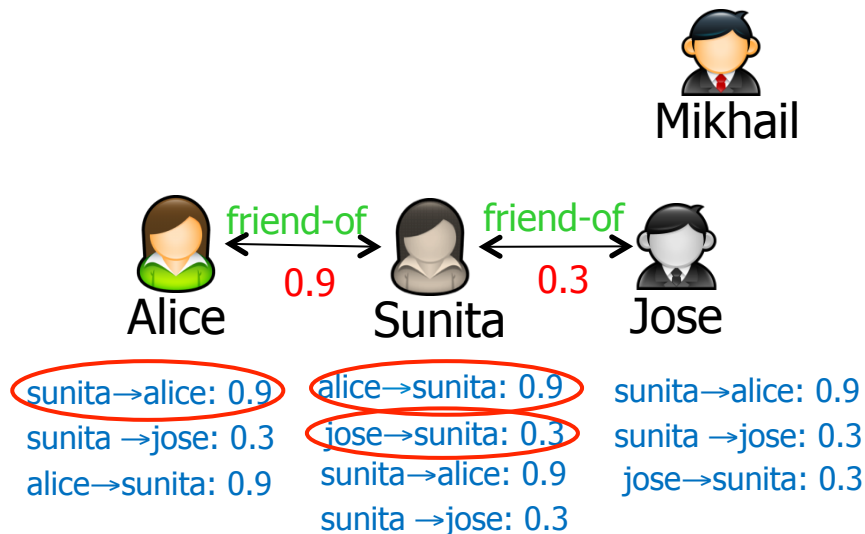
- Example: Am I my friends' best friend?
 - Step #1: Discard irrelevant vertices and edges
 - Step #2: Annotate each vertex with list of friends
 - Step #3: Push annotations along each edge

A computation model for graphs



- Example: Am I my friends' best friend?
 - Step #1: Discard irrelevant vertices and edges
 - Step #2: Annotate each vertex with list of friends
 - Step #3: Push annotations along each edge

A computation model for graphs



- Example: Am I my friends' best friend?
 - Step #1: Discard irrelevant vertices and edges
 - Step #2: Annotate each vertex with list of friends
 - Step #3: Push annotations along each edge
 - Step #4: Determine result at each vertex

Can we do this in MapReduce?

```
map(key: node, value: [<otherNode, relType, strength>])
{
  for each otherNode, relType, strength in value:
    for each n in list of otherNodes:
      emit(n, <node, otherNode, relType, strength>)
}
reduce(key: node, values: list of <src, dst, relType, strength>)
{
  for each <src,dst> find the dst where strength is highest:
    if (node != dst) emit(node, "NO"); return
  emit(node, "YES")
}
```

- Using adjacency list representation?

Can we do this in MapReduce?

```
map(key: node, value: <otherNode, relType, strength>)  
{  
  
}  
reduce(key: _____, values: list of _____)  
{  
  
}
```

- Using single-edge data representation?

A real-world use case

- A variant that is actually used in social networks today: "Who are the friends of multiple of my friends?"
 - Where have you seen this before?
- Friend recommendation!
 - Maybe these people should be my friends too!

Generalizing...

- Now suppose we want to go beyond direct friend relationships
 - Example: How many of my friends' friends (distance-2 neighbors) have me as their best friend's best friend?
 - What do we need to do?
- How about distance $k > 2$?
- To compute the answer, we need to run multiple iterations of MapReduce!

Iterative MapReduce

- The basic model:

```
copy files from input dir → staging dir N = 1  
(optional: do some preprocessing)
```

```
while (!terminating condition) {  
    map from staging dir N  
    reduce into staging dir N+1  
    N = N + 1 (optimization: use only 2 staging dirs alternately)  
}
```

```
(optional: postprocessing)
```

```
move or process files from staging dir N+1 → output dir
```

- Note that reduce output must be compatible with the map input!
 - What can happen if we filter out some information in the mapper or in the reducer?


Graph algorithms and MapReduce

- A centralized algorithm typically traverses a tree or a graph one item at a time (there's only one "cursor")
 - You've learned breadth-first and depth-first traversals
- Most algorithms that are based on graphs make use of multiple map/reduce stages processing one "wave" at a time

Recap: MapReduce on graphs

- Suppose we want to:
 - compute a function for each vertex in a graph...
 - ... using data from vertices at most k hops away
- We can do this as follows:
 - "Push" information along the edges
 - "Think like a vertex"
 - Finally, perform the computation at each vertex
- May need more than one MapReduce phase
 - Iterative MapReduce: Outputs of stage $i \rightarrow$ inputs of stage $i+1$

Plan for today

- Representing data in graphs ✓
- Graph algorithms in MapReduce ✓
 - Computation model ✓
 - Iterative MapReduce ✓
- A toolbox of algorithms 
 - Single-source shortest path (SSSP)
 - k-means clustering
 - Classification with Naïve Bayes
 - PageRank

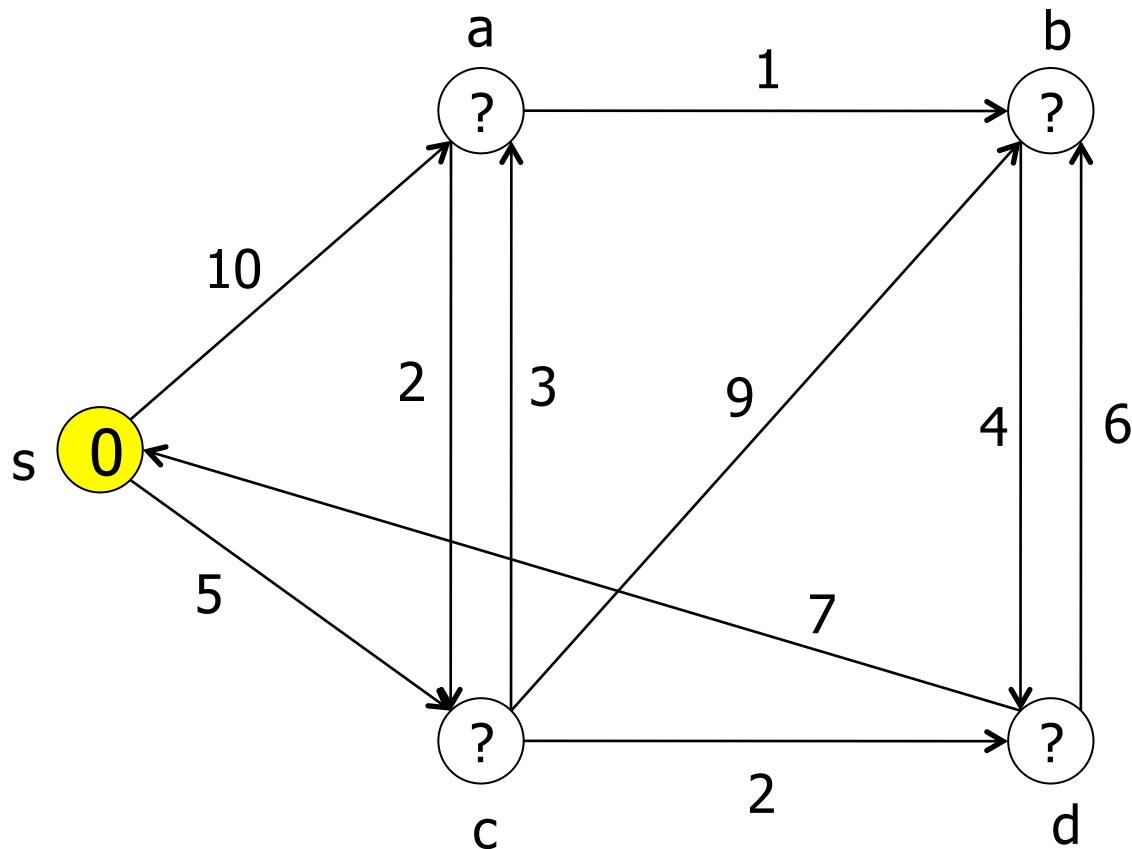
Path-based algorithms

- Sometimes our goal is to compute information about the paths (sets of paths) between nodes
 - Edges may be annotated with **cost**, **distance**, or **similarity**
- Examples of such problems:
 - **Shortest path** from one node to another
 - Minimum spanning tree (minimal-cost tree connecting all vertices in a graph)
 - Steiner tree (minimal-cost tree connecting certain nodes)
 - Topological sort (node in a DAG comes before all nodes it points to)

Single-Source Shortest Path (SSSP)

Given a directed graph $G = (V, E)$ in which each edge e has a cost $c(e)$:

- Compute the cost of reaching each node from the source node s in the **most efficient way** (potentially after multiple 'hops')



SSSP: Intuition

- We can formulate the problem using induction
 - The shortest path follows the **principle of optimality**: the last step (u,v) makes use of the shortest path to u
- We can express this as follows:

```
bestDistanceAndPath(v) {  
  if (v == source) then {  
    return <distance 0, path[v]>  
  } else {  
    find argmin_u (bestDistanceAndPath[u] + dist[u,v])  
    return <bestDistanceAndPath[u] + dist[u,v], path[u] + v>  
  }  
}
```

SSSP: Traditional Solution

■ Dijkstra's algorithm

`V: vertices, E: edges, S: start node`

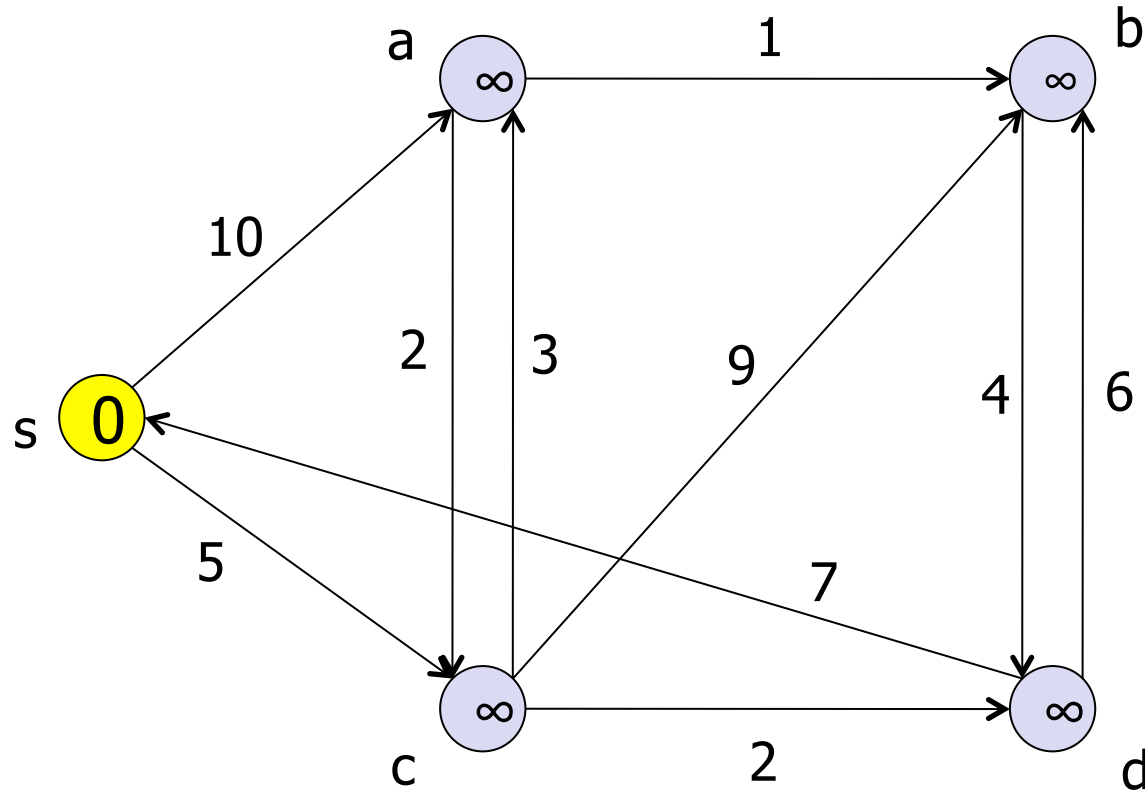
```
foreach v in V
    dist_S_To[v] := infinity
    predecessor[v] = nil
dist_S_To[S] := 0
Q := V
```

Initialize length and
last step of path
to default values

```
while (Q not empty) do
    u := Q.removeNodeClosestTo(S)
    foreach v in V where (u,v) in E
        if (dist_S_To[v] > dist_S_To[u] + cost(u,v)) then
            dist_S_To[v] = dist_S_To[u] + cost(u,v)
            predecessor[v] = u
```

Update length and
path based on edges
radiating from u

SSSP: Dijkstra in Action

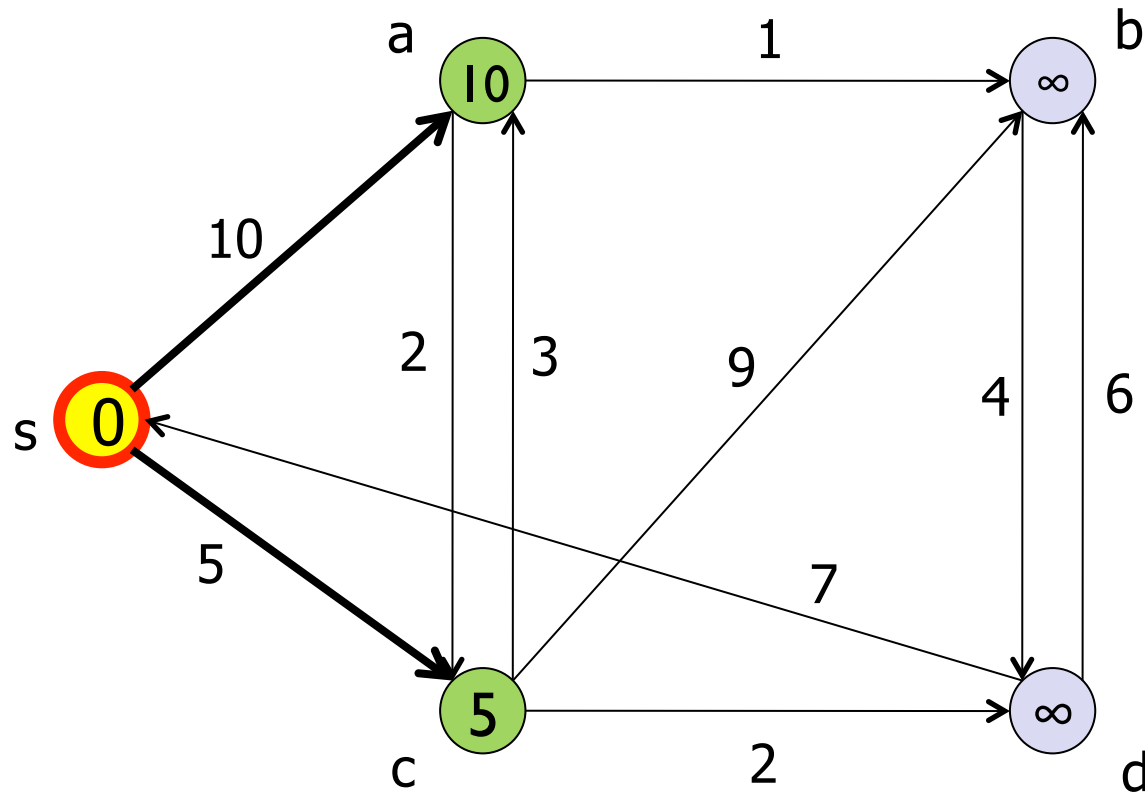


$Q = \{s, a, b, c, d\}$

$\text{dist_S_To}: \{(a, \infty), (b, \infty), (c, \infty), (d, \infty)\}$

$\text{predecessor}: \{(a, \text{nil}), (b, \text{nil}), (c, \text{nil}), (d, \text{nil})\}$

SSSP: Dijkstra in Action

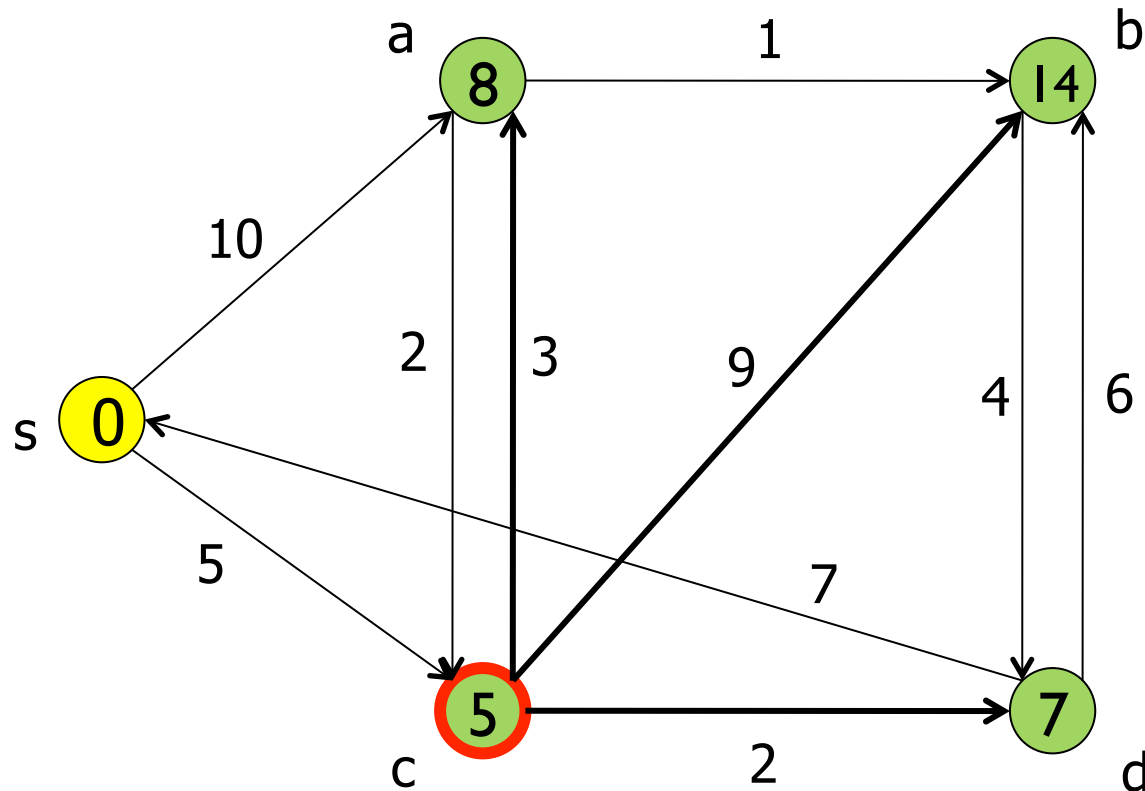


$Q = \{a, b, c, d\}$

dist_S_To: $\{(a, 10), (b, \infty), (c, 5), (d, \infty)\}$

predecessor: $\{(a, s), (b, \text{nil}), (c, s), (d, \text{nil})\}$

SSSP: Dijkstra in Action

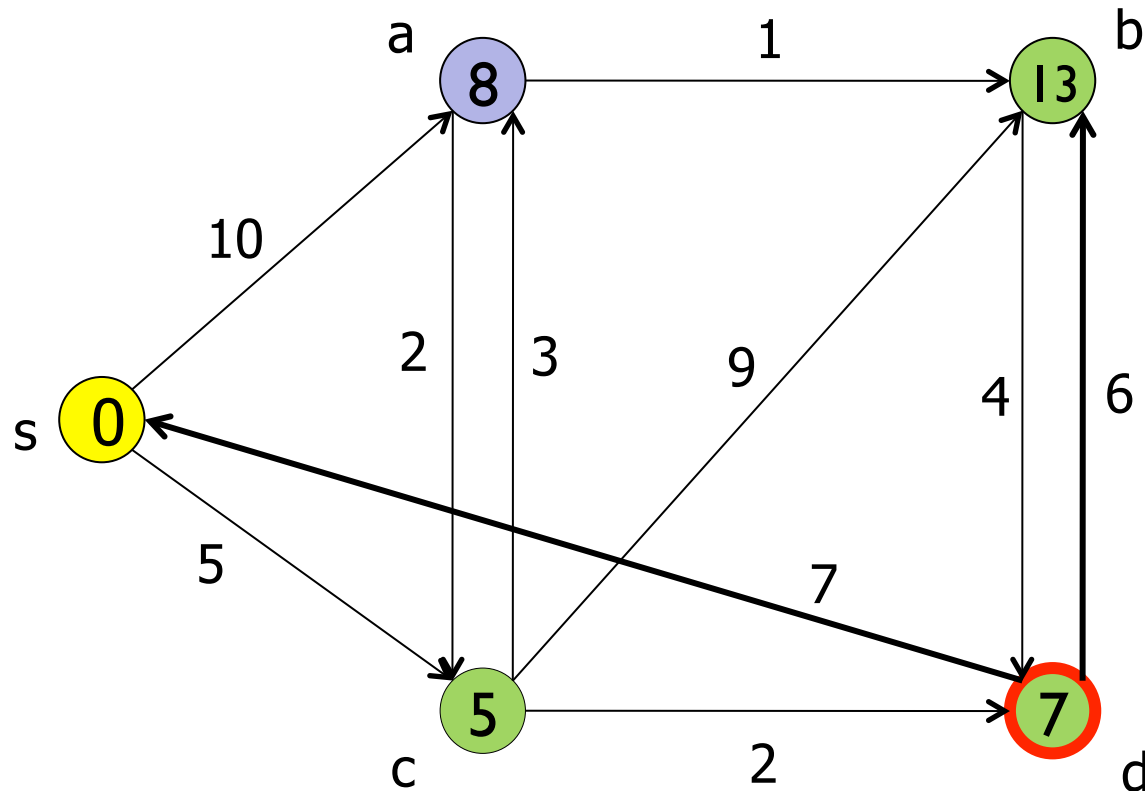


$Q = \{a, b, d\}$

$\text{dist_S_To} = \{(a, 8), (b, 14), (c, 5), (d, 7)\}$

$\text{predecessor} = \{(a, c), (b, c), (c, s), (d, c)\}$

SSSP: Dijkstra in Action

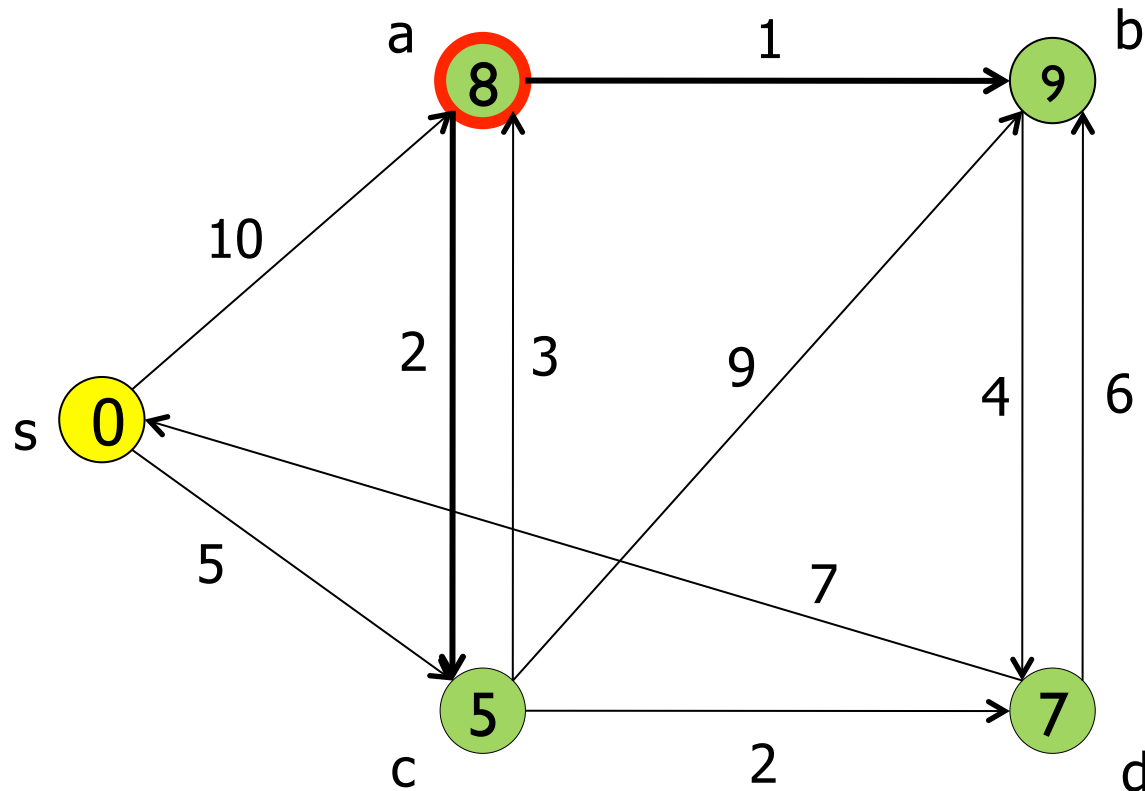


$Q = \{a, b\}$

dist_S_To: $\{(a, 8), (b, 13), (c, 5), (d, 7)\}$

predecessor: $\{(a, c), (b, d), (c, s), (d, c)\}$

SSSP: Dijkstra in Action

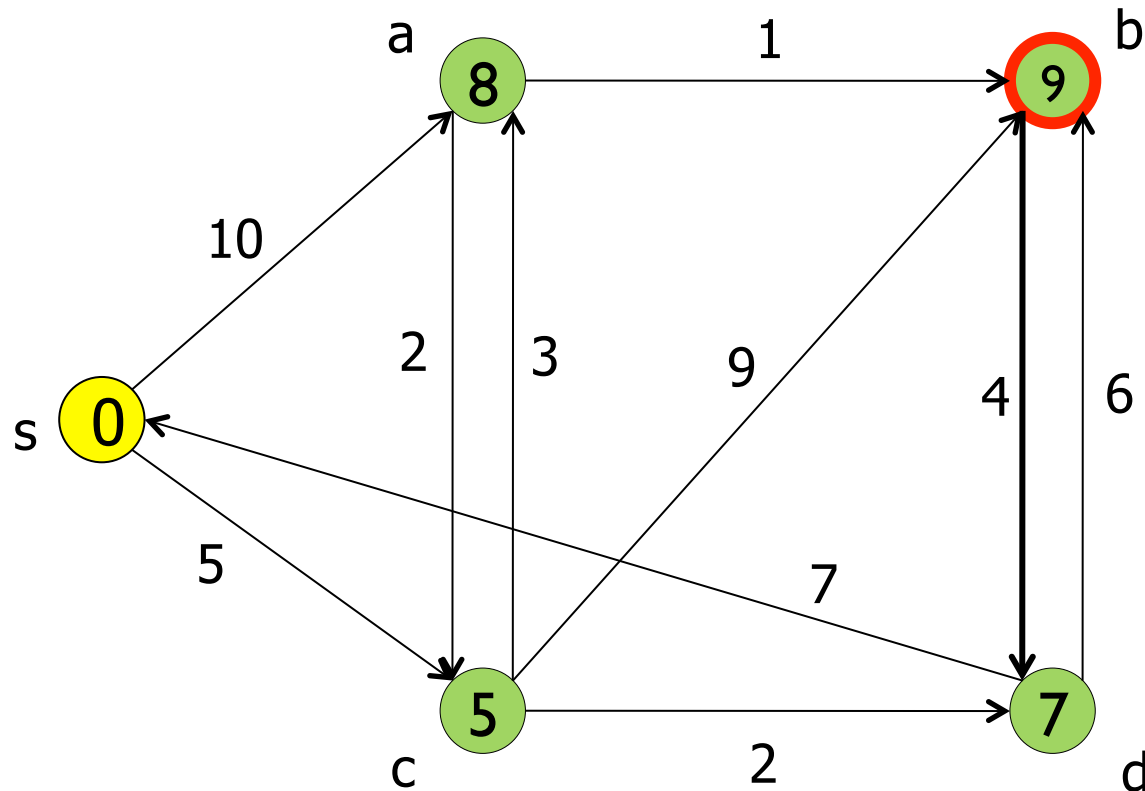


$Q = \{b\}$

dist_S_To: $\{(a,8), (b,9), (c,5), (d,7)\}$

predecessor: $\{(a,c), (b,a), (c,s), (d,c)\}$

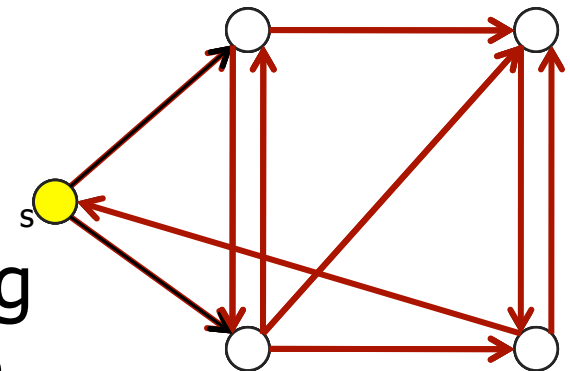
SSSP: Dijkstra in Action



$Q = \{\}$
 $\text{dist_S_To: } \{(a,8), (b,9), (c,5), (d,7)\}$
 $\text{predecessor: } \{(a,c), (b,a), (c,s), (d,c)\}$

SSSP: How to parallelize?

- Dijkstra traverses the graph along a single route at a time, prioritizing its traversal to the next step based on total path length (and avoiding cycles)
 - No real parallelism to be had here!
- Intuitively, we want something that “radiates” from the origin, one “edge hop distance” at a time
 - Each step outwards can be done in parallel, before another iteration occurs - or we are done
 - Scalability depends on the algorithm, not (just) on the problem!



SSSP: Revisiting the inductive definition

```
bestDistanceAndPath(v) {  
  if (v == source) then {  
    return <distance 0, path [v]>  
  } else {  
    find argmin_u (bestDistanceAndPath[u] + dist[u,v])  
    return <bestDistanceAndPath[u] + dist[u,v], path[u] + v>  
  }  
}
```

- Dijkstra's algorithm carefully considered each u in a way that allowed us to **prune** certain points
- Instead we can look at all potential u 's for each v
 - Compute iteratively, by keeping a "frontier set" of u nodes i edge-hops from the source

SSSP: MapReduce formulation

■ init:

- For each node, node ID $\rightarrow \langle \infty, -, \{ \langle \text{succ-node-ID}, \text{edge-cost} \rangle \} \rangle$
- The shortest path we have found so far from the source to nodeID has length ∞ ...
... this is the next hop on that path...
... and here is the adjacency list for nodeID

■ map:

- take node ID $\rightarrow \langle \text{distance}, \text{next}, \{ \langle \text{succ-node-ID}, \text{edge-cost} \rangle \} \rangle$
 - For each succ-node-ID:
 - emit succ-node ID $\rightarrow \{ \langle \text{node ID}, \text{distance} + \text{edge-cost} \rangle \}$
 - emit node ID $\rightarrow \text{distance}, \{ \langle \text{succ-node-ID}, \text{edge-cost} \rangle \}$
- This is a new path from the source to succ-node-ID that we just discovered (not necessarily shortest)

■ reduce:

- distance := min cost from a predecessor; next := that predec.
 - emit node ID $\rightarrow \langle \text{distance}, \text{next}, \{ \langle \text{succ-node-ID}, \text{edge-cost} \rangle \} \rangle$
- Why is this necessary?

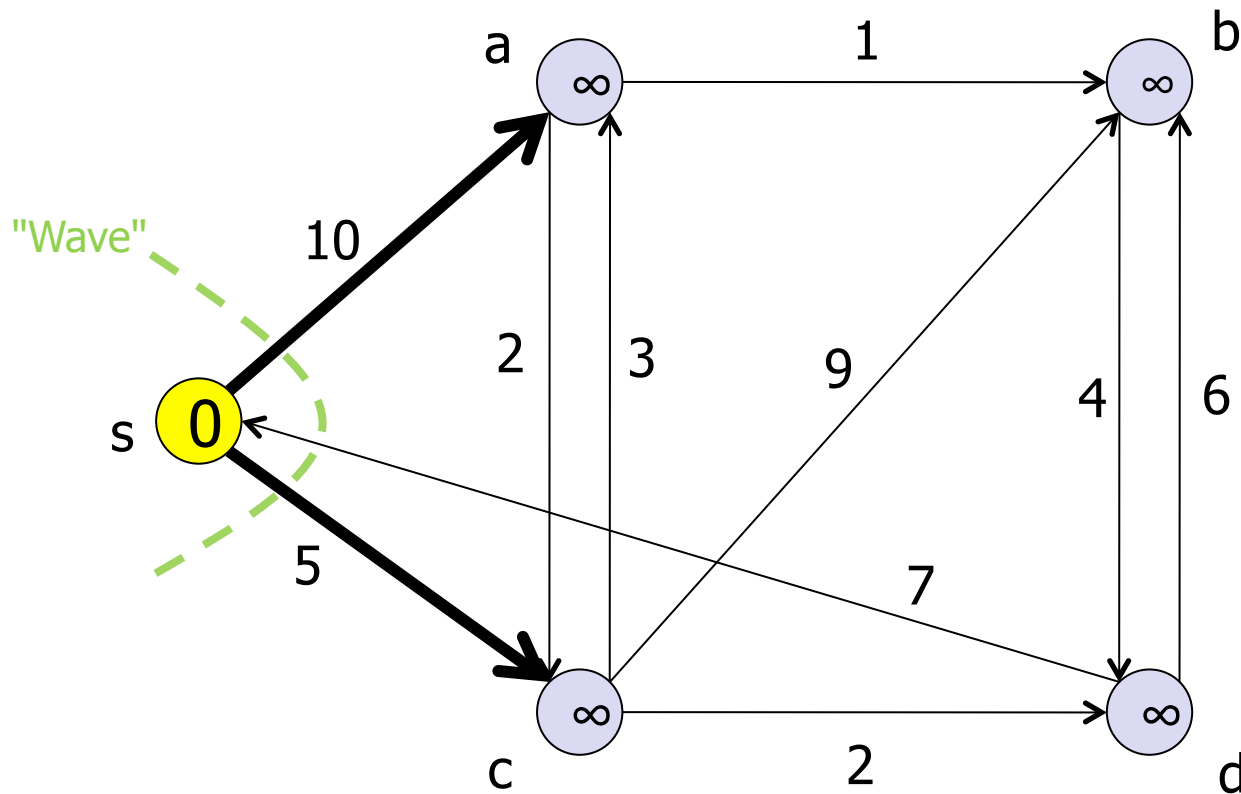
■ Repeat until no changes

■ Postprocessing: Remove adjacency lists

Iteration 0: Base case

mapper: $(a, \langle s, 10 \rangle)$ $(c, \langle s, 5 \rangle)$ edges

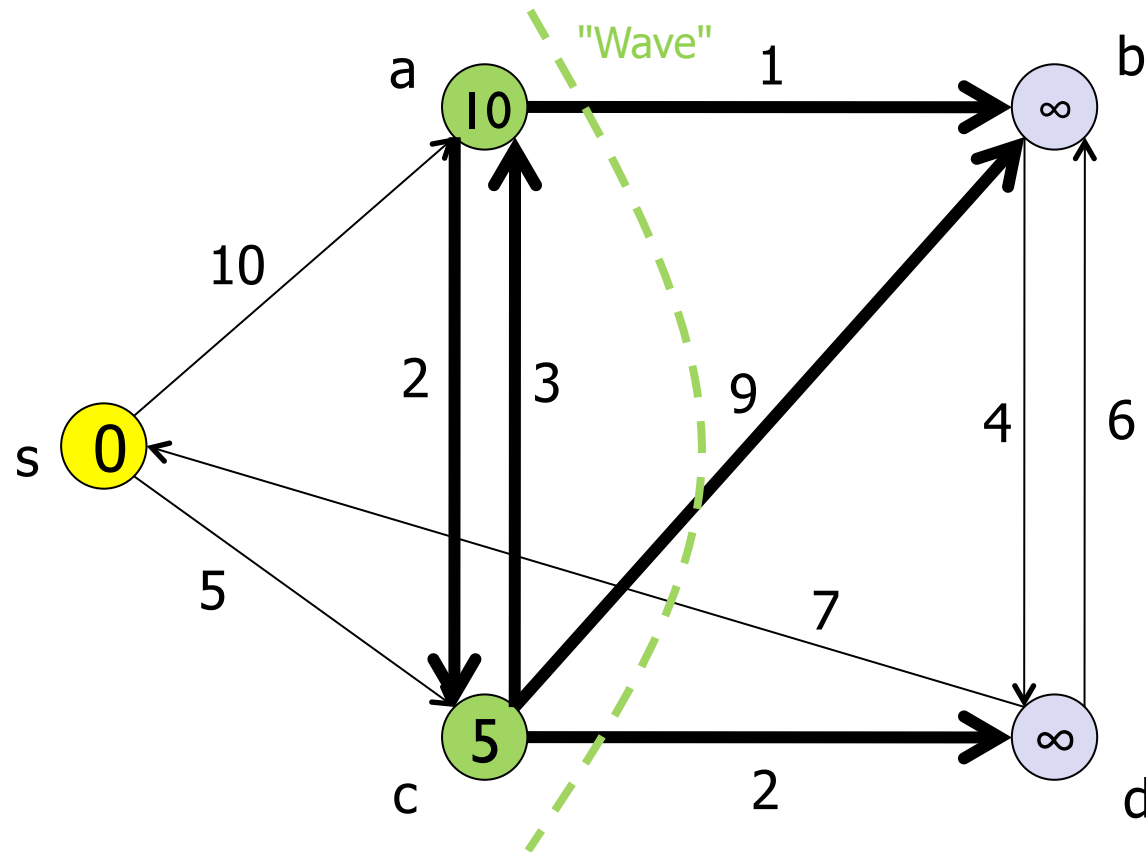
reducer: $(a, \langle 10, \dots \rangle)$ $(c, \langle 5, \dots \rangle)$



Iteration 1

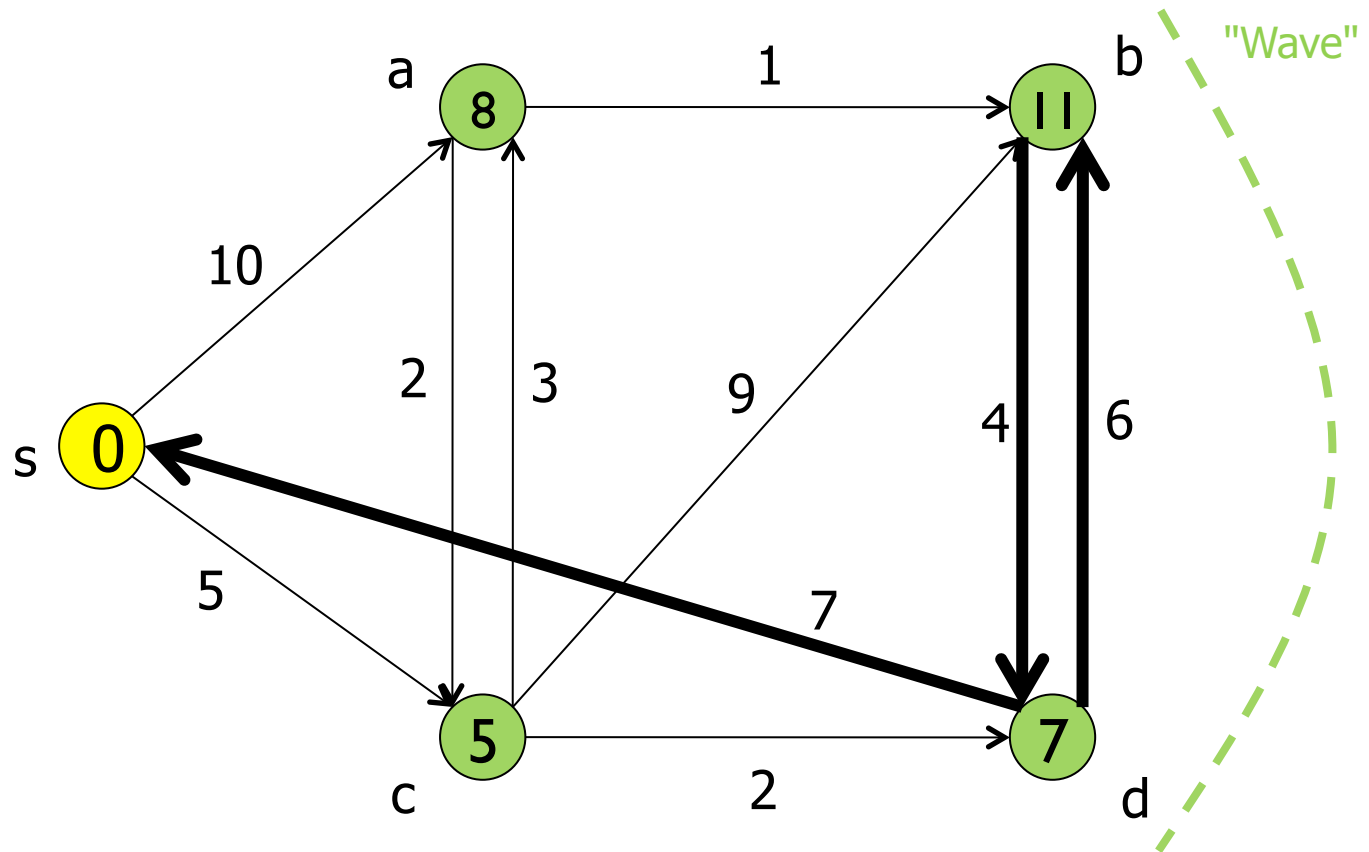
mapper: (a,<s,10>) (c,<s,5>) (a,<c,8>) (c,<a,9>) (b,<a,11>)
(b,<c,14>) (d,<c,7>) edges

reducer: (a,<8, ...>) (c,<5, ...>) (b,<11, ...>) (d,<7, ...>)



Iteration 2

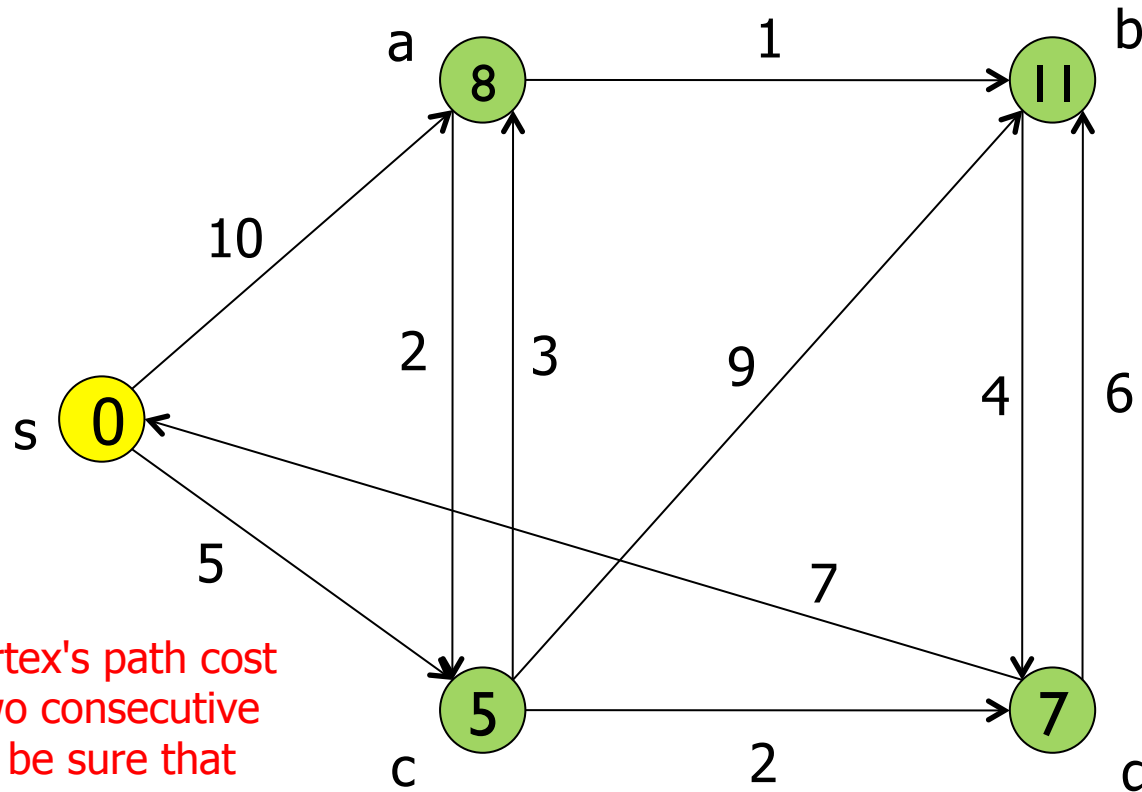
mapper: (a,<s,10>) (c,<s,5>) (a,<c,8>) (c,<a,9>) (b,<a,11>)
(b,<c,14>) (d,<c,7>) (b,<d,13>) (d,<b,15>) edges
reducer: (a,<8>) (c,<5>) (b,<11>) (d,<7>)



Iteration 3

No change!
Convergence!

mapper: (a,<s,10>) (c,<s,5>) (a,<c,8>) (c,<a,9>) (b,<a,11>)
(b,<c,14>) (d,<c,7>) (b,<d,13>) (d,<b,15>) edges
reducer: (a,<8>) (c,<5>) (b,<11>) (d,<7>)



Question: If a vertex's path cost is the same in two consecutive rounds, can we be sure that this vertex has converged?

Summary: SSSP

- Path-based algorithms typically involve iterative map/reduce
- They are typically formulated in a way that traverses in “waves” or “stages”, like breadth-first search
 - This allows for parallelism
 - They need a way to test for convergence
- Example: Single-source shortest path (SSSP)
 - Original Dijkstra formulation is hard to parallelize
 - But we can make it work with the “wave” approach

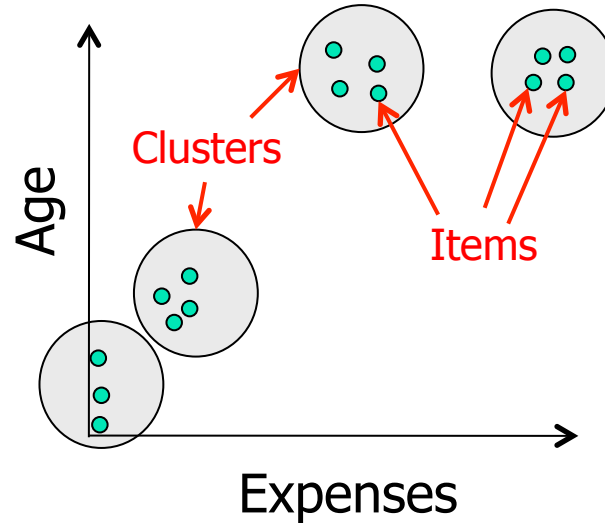
Plan for today

- Representing data in graphs ✓
- Graph algorithms in MapReduce ✓
 - Computation model ✓
 - Iterative MapReduce ✓
- A toolbox of algorithms
 - Single-source shortest path (SSSP) ✓
 - k-means clustering ← NEXT
 - Classification with Naïve Bayes
 - PageRank

Learning (clustering / classification)

- Sometimes our goal is to take a set of entities, possibly related, and group them
 - If the groups are based on similarity, we call this **clustering**
 - If the groups are based on putting them into a semantically meaningful class, we call this **classification**
- Both are instances of **machine learning**

The k-clustering Problem



- Given: A set of items in a n -dimensional **feature space**
 - Example: data points from survey, people in a social network
- Goal: Group the items into k “clusters”
 - What would be a 'good' set of clusters?

Approach: k-Means

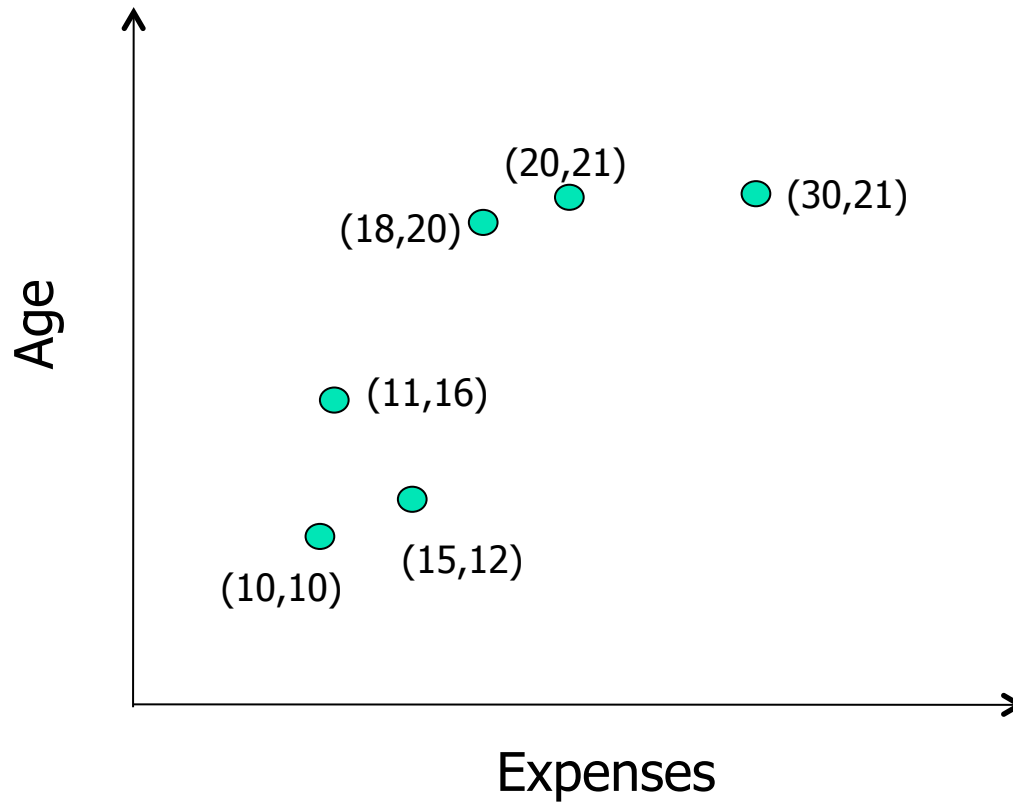
- Let m_1, m_2, \dots, m_k be representative points for each of our k clusters
 - Specifically: the **centroid** of the cluster
- Initialize m_1, m_2, \dots, m_k to random values in the data
- For $t = 1, 2, \dots$:
 - Assign each point to the closest centroid

$$S_i^{(t)} = \{x_j : \|x_j - m_i^{(t)}\| \leq \|x_j - m_{i^*}^{(t)}\|, i^* = 1, \dots, k\}$$

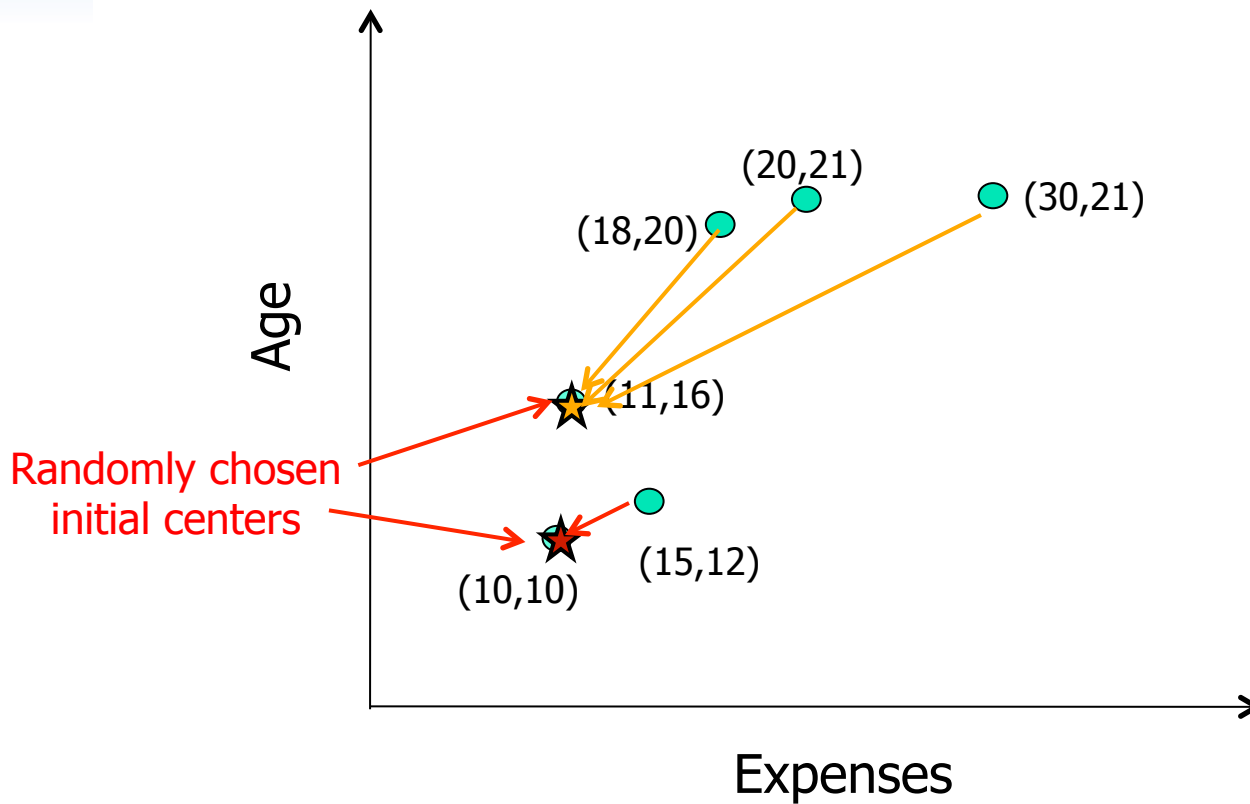
- m_i becomes new centroid for its points

$$m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$$

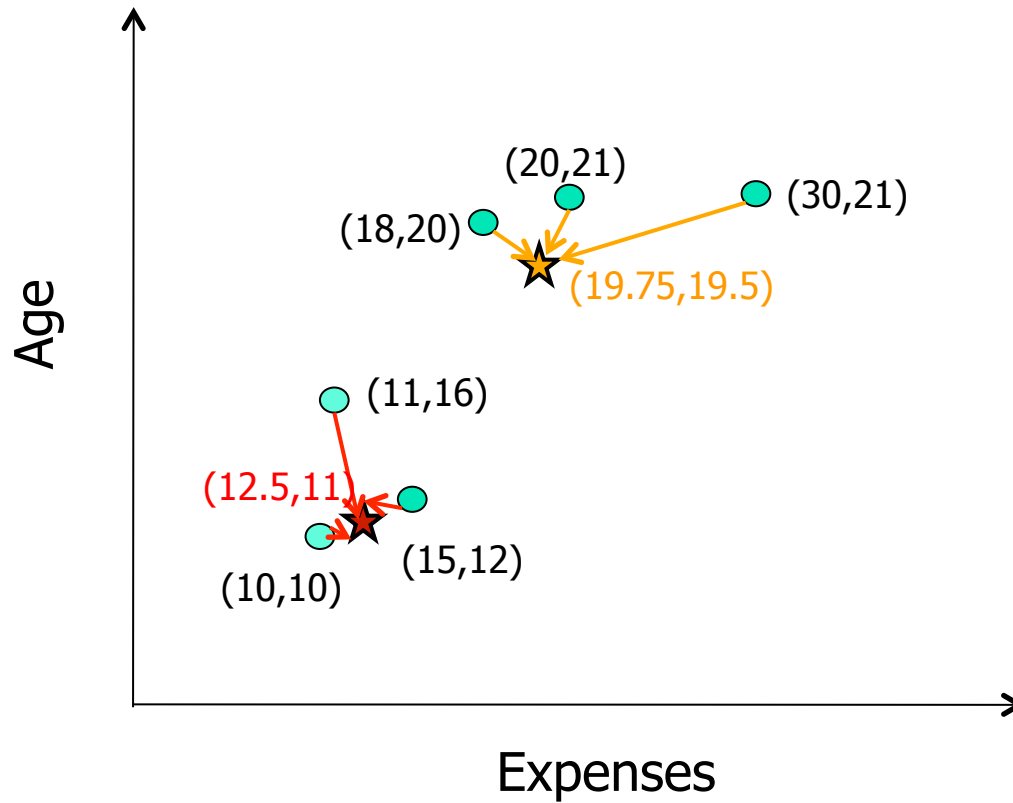
A simple example (1/4)



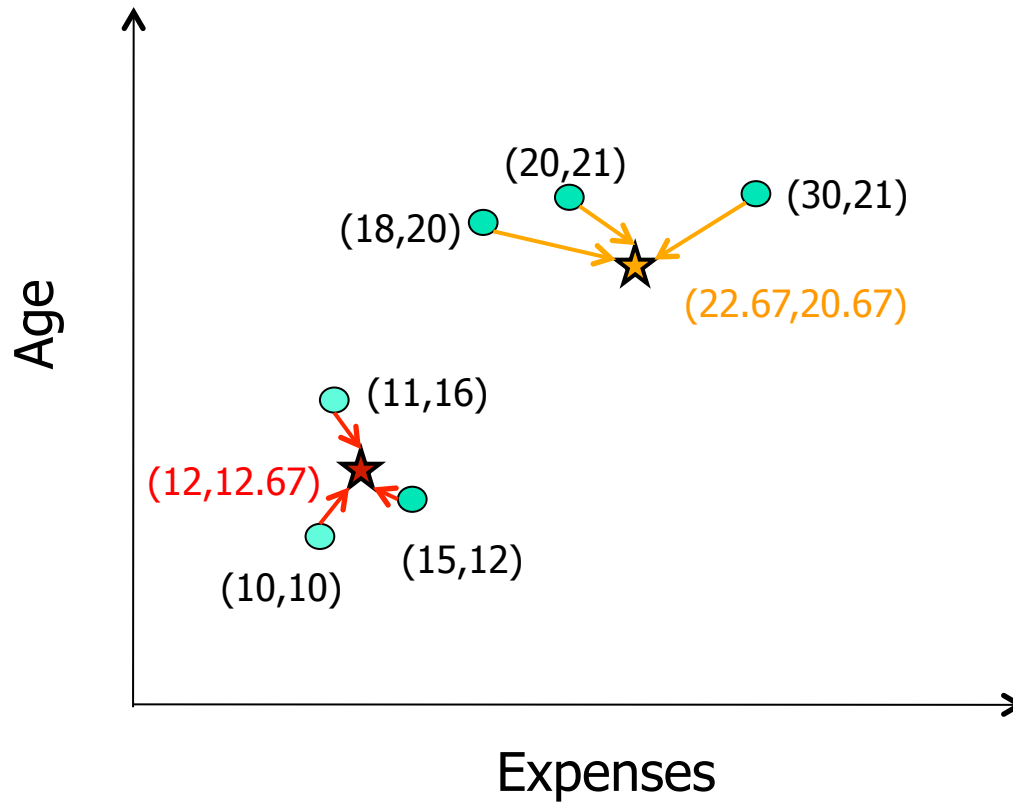
A simple example (2/4)



A simple example (3/4)



A simple example (4/4)



Stable!

k-Means in MapReduce

■ Classify

- Assign each point to the closest centroid

$$S_i^{(t)} = \left\{ x_j : \|x_j - m_i^{(t)}\| \leq \|x_j - m_{i^*}^{(t)}\|, i^* = 1, \dots, k \right\}$$

This is the Map phase



■ Recenter

- m_i becomes new centroid for its points

$$m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$$

This is the Reduce phase



■ Repeat until no change

- Centroids have converged

Classification Step as Map

- **init:**

- Read in global var centroids from file
 - Initially k random points

- **map(centroid, point):**

- Compute nearest centroid based on centroids
- emit(nearest centroid, point)

How do we know the centroids?



Recenter Step as Reduce

- Initialize global var centroids = []
- `reduce(centroid, points[])`
 - Recompute centroid from points in it
 - Foreach point in points:
 - `emit(centroid, point)`
 - Add centroid to global centroids
- cleanup (after all calls to reduce are made):
 - Save global centroids to file

Practical Notes

- After reduce phase finishes, must check if any centroid has changed and, in such case, start another MapReduce iteration
- Our solution doesn't fit pure MapReduce model
 - It uses global variable to efficiently know current centroids
 - This state is read and written to shared file
 - When more than 1 reducer is used, it needs to avoid file conflicts and merge before next iteration of MapReduce
- Can you think of solution without global state?
 - What data needs to be passed around?
 - How many MapReduce jobs?

Plan for today

- Representing data in graphs ✓
- Graph algorithms in MapReduce ✓
 - Computation model ✓
 - Iterative MapReduce ✓
- A toolbox of algorithms
 - Single-source shortest path (SSSP) ✓
 - k-means clustering ✓
 - Classification with Naïve Bayes
 - PageRank



Classification

- Suppose we want to learn what is spam (or interesting, or ...)
 - Predefine a set of **classes** with semantic meaning
 - **Train** an algorithm to look at data and assign a class
 - Based on giving it some **examples** of data in each class
 - ... and the sets of **features** they have
- Many probabilistic techniques exist
 - Each class has probabilistic relationships with others
 - e.g., $p(\text{spam} \mid \text{isSentLocally})$, $p(\text{isSentLocally} \mid \text{fromBob})$, ...
 - But we'll focus on a simple, "flat" model: Naïve Bayes



A simple example

- Suppose we just look at the keywords in the email's title:

Message(1, "Won contract")

Message(2, "Won award")

Message(3, "Won the lottery")

Message(4, "Unsubscribe")

Message(5, "Millions of customers")

Message(6, "Millions of dollars")



- What is **probability** message "Won Millions" is  ?

$p(\text{spam} \mid \text{containsWon}, \text{containsMillions})$

$$= \frac{p(\text{spam}) p(\text{containsWon}, \text{containsMillions} \mid \text{spam})}{p(\text{containsWon}, \text{containsMillions})}$$

Bayes' Theorem

Classification using Naïve Bayes

- Basic assumption: Probabilities of events are independent
 - This is why it is called 'naïve'
- Under this assumption,

$$\frac{p(\text{spam}) p(\text{containsWon,containsMillions} \mid \text{spam})}{p(\text{containsWon,containsMillions})}$$

$$= \frac{p(\text{spam}) p(\text{containsWon} \mid \text{spam}) p(\text{containsMillions} \mid \text{spam})}{p(\text{containsWon}) p(\text{containsMillions})}$$

$$= 0.5 * 0.67 * 0.33 / (0.5 * 0.33) = 0.67$$

- So how do we “train” a learner (compute the above probabilities) using MapReduce?

What do we need to train the learner?

- $p(\text{spam})$

- Count how many spam emails there are
- Count total number of emails

Easy

Easy

- $p(\text{containsXYZ} \mid \text{spam})$

- Count how many spam emails contain XYZ
- Count how many emails contain XYZ overall

1

2

- $p(\text{containsXYZ})$

- Count how many emails contain XYZ overall
- Count total number of emails

2

Easy

Training a Naïve Bayes Learner

■ map 1:

- takes messageId \rightarrow $\langle \text{class}, \{\text{words}\} \rangle$
- emits $\langle \text{word}, \text{class} \rangle \rightarrow 1$

■ reduce 1:

- emits $\langle \text{word}, \text{class} \rangle \rightarrow \langle \text{count} \rangle$

Count how many
emails in the class
contain the word
(modified WordCount)

■ map 2:

- takes messageId \rightarrow $\langle \text{class}, \{\text{words}\} \rangle$
- emits word $\rightarrow 1$

■ reduce 2:


- emits word $\rightarrow \langle \text{totalCount} \rangle$

Count how many
emails contain the
word overall
(WordCount)

Summary: Learning and MapReduce

- Clustering algorithms typically have multiple aggregation stages or iterations
 - k-means clustering repeatedly computes centroids, maps items to them
 - Fixpoint computation
- Classification algorithms can be quite complex
 - In general: need to capture conditional probabilities
 - Naïve Bayes assumes everything is independent
 - Training is a matter of computing probability distribution
 - Can be accomplished using two Map/Reduce passes

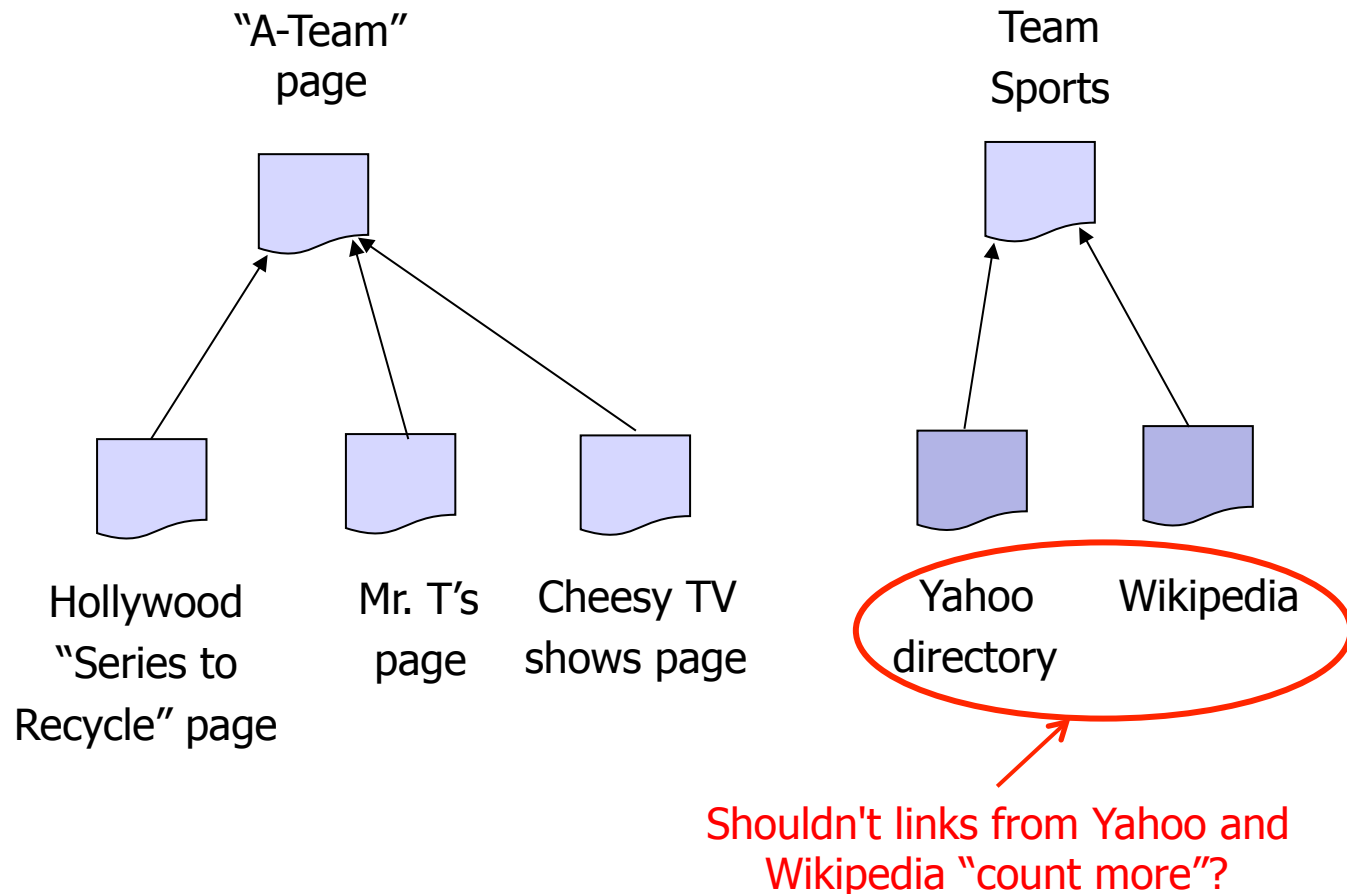
Plan for today

- Representing data in graphs ✓
- Graph algorithms in MapReduce ✓
 - Computation model ✓
 - Iterative MapReduce ✓
- A toolbox of algorithms
 - Single-source shortest path (SSSP) ✓
 - k-means clustering ✓
 - Classification with Naïve Bayes ✓
 - PageRank 

Why link analysis?

- Suppose a search engine processes a query for "team sports"
 - Problem: Millions of pages contain these words!
 - Which ones should we return first?
- **Idea:** Hyperlinks encode a considerable amount of human judgment
 - What does it mean when a web page links another page?
 - Intra-domain links: Often created primarily for navigation
 - Inter-domain links: Confer some measure of authority
- So, can we simply boost the rank of pages with lots of inbound links?

Problem: Popularity \neq relevance!

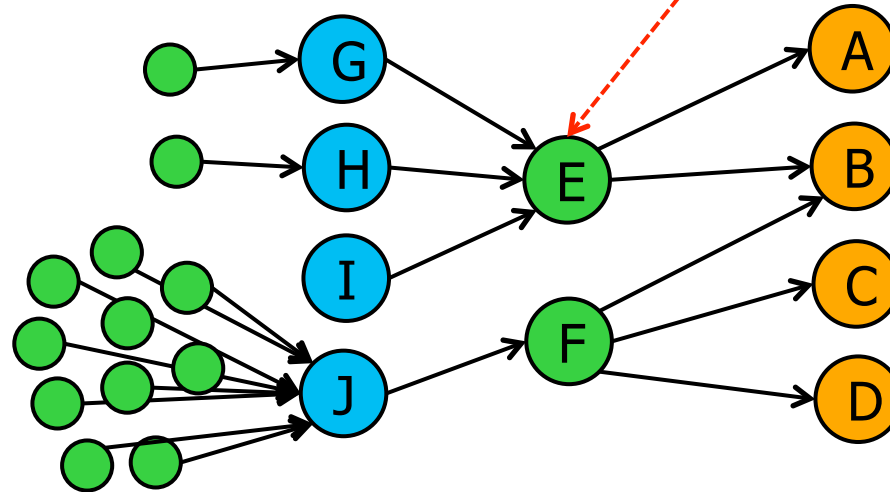


Other applications

- This question occurs in several other areas:
 - Who are the most "influential" individuals in a social network? (#friends?)
 - How do we measure the "impact" of a researcher? (#papers? #citations?)
 - Which programmers are writing the "best" code? (#uses?)
 - ...

PageRank: Intuition

Shouldn't E's vote be worth more than F's?



- Imagine a contest for The Web's Best Page
 - Initially, each page has one vote
 - Each page votes for all the pages it has a link to
 - To ensure fairness, pages voting for more than one page must split their vote equally between them
 - Voting proceeds in rounds; in each round, each page has the number of votes it received in the previous round
 - In practice, it's a little more complicated - but not much!

PageRank

- Each page i is given a rank x_i
- Goal: Assign the x_i such that the rank of each page is governed by the ranks of the pages linking to it:

$$x_i = \sum_{j \in B_i} \frac{1}{N_j} x_j$$

Rank of page i

Rank of page j

Number of links out from page j

Every page j that links to i

How do we compute the rank values?

Random Surfer Model

- PageRank has an intuitive basis in random walks on graphs
- Imagine a **random surfer**, who starts on a random page and, in each step,
 - with probability d , clicks on a random link on the page
 - with probability $1-d$, jumps to a random page (bored?)
- The PageRank of a page can be interpreted as the fraction of steps the surfer spends on the corresponding page
 - Transition matrix can be interpreted as a Markov Chain

Iterative PageRank (simplified)

Initialize all ranks to
be equal, e.g.:

$$x_i^{(0)} = \frac{1}{n}$$

Iterate until
convergence

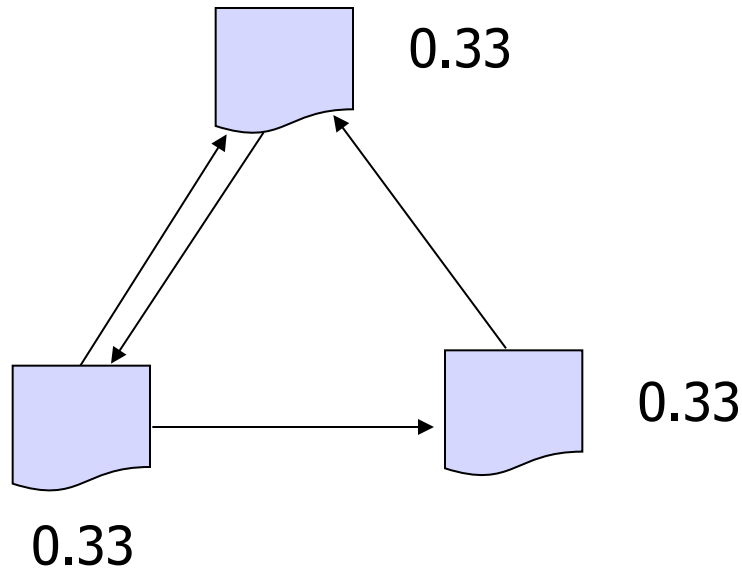
$$x_i^{(k+1)} = \sum_{j \in B_i} \frac{1}{N_j} x_j^{(k)}$$

No need to decide
how many levels
to consider!

Example: Step 0

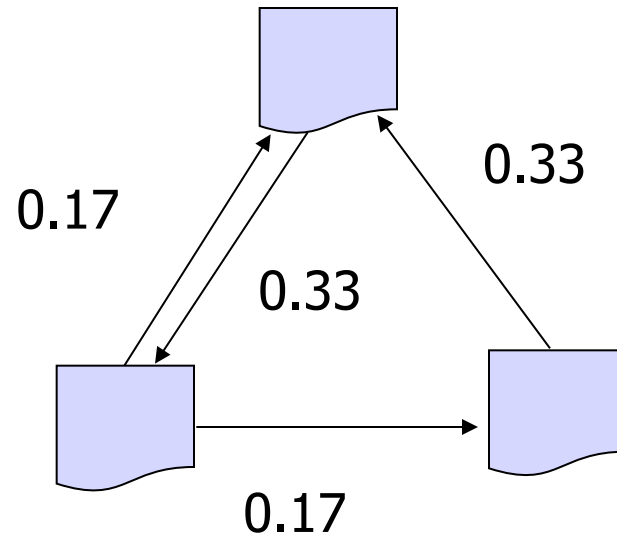
Initialize all ranks
to be equal

$$x_i^{(0)} = \frac{1}{n}$$



Example: Step 1

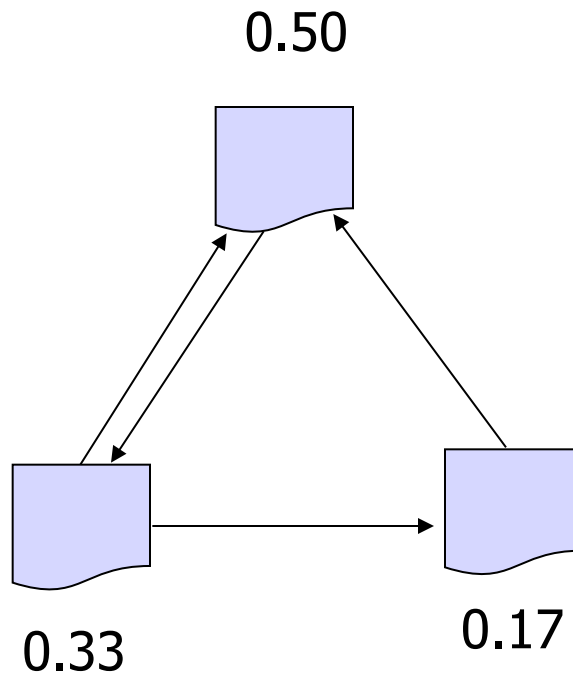
Propagate weights
across out-edges



Example: Step 2

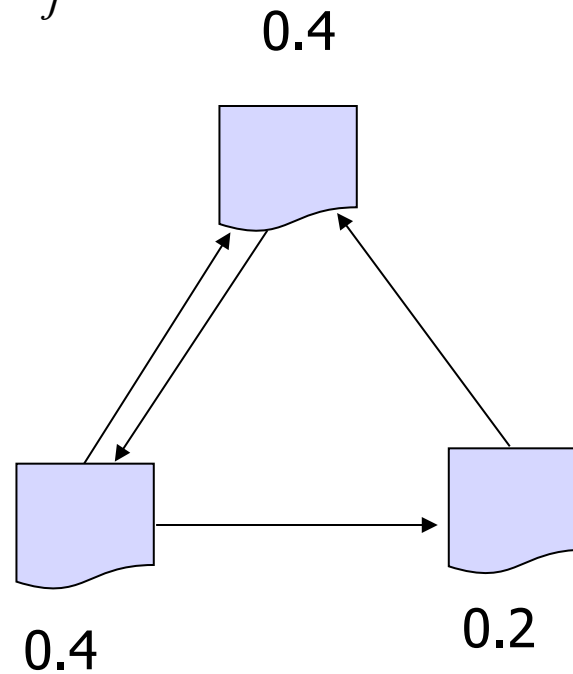
Compute weights
based on in-edges

$$x_i^{(1)} = \sum_{j \in B_i} \frac{1}{N_j} x_j^{(0)}$$



Example: Convergence

$$x_i^{(k+1)} = \sum_{j \in B_i} \frac{1}{N_j} x_j^{(k)}$$



Naïve PageRank Algorithm Restated

- Let

- $N(p)$ = number outgoing links from page p
- $B(p)$ = number of back-links to page p

$$PageRank(p) = \sum_{b \in B(p)} \frac{1}{N(b)} PageRank(b)$$

- Each page b distributes its importance to all of the pages it points to (so we scale by $1/N(b)$)
- Page p 's importance is increased by the importance of its back set

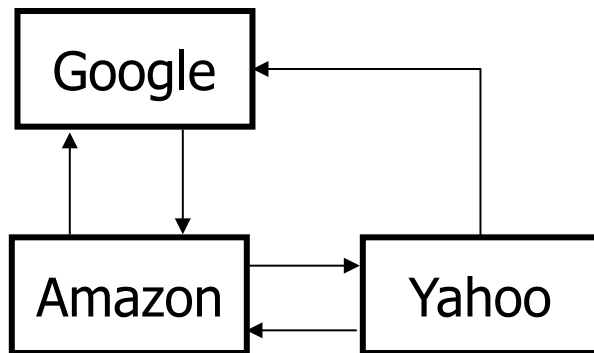
In Linear Algebra formulation

- Create an $m \times m$ matrix M to capture links:
 - $M(i, j) = 1 / n_j$ if page i is pointed to by page j
and page j has n_j outgoing links
 $= 0$ otherwise
- Initialize all PageRanks to 1, multiply by M repeatedly until all values converge:

$$\begin{bmatrix} \text{PageRank}(p_1') \\ \text{PageRank}(p_2') \\ \dots \\ \text{PageRank}(p_m') \end{bmatrix} = M \begin{bmatrix} \text{PageRank}(p_1) \\ \text{PageRank}(p_2) \\ \dots \\ \text{PageRank}(p_m) \end{bmatrix}$$

- Computes **principal eigenvector** via **power iteration**

A brief example



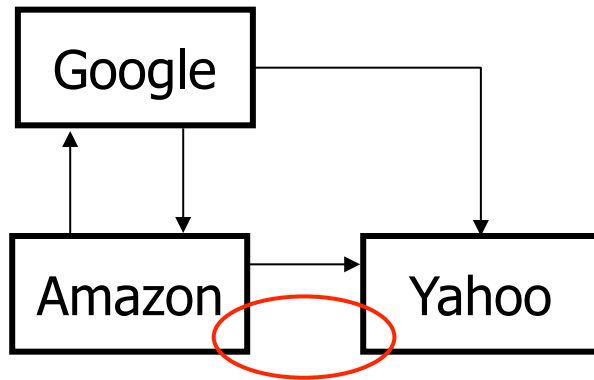
$$\begin{bmatrix} g' \\ y' \\ a' \end{bmatrix} = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0 & 0 & 0.5 \\ 1 & 0.5 & 0 \end{bmatrix} * \begin{bmatrix} g \\ y \\ a \end{bmatrix}$$

Running for multiple iterations:

$$\begin{bmatrix} g \\ y \\ a \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0.5 \\ 1.5 \end{bmatrix}, \begin{bmatrix} 1 \\ 0.75 \\ 1.25 \end{bmatrix}, \dots \begin{bmatrix} 1 \\ 0.67 \\ 1.33 \end{bmatrix}$$

Total rank sums to number of pages

Oops #1 – PageRank sinks



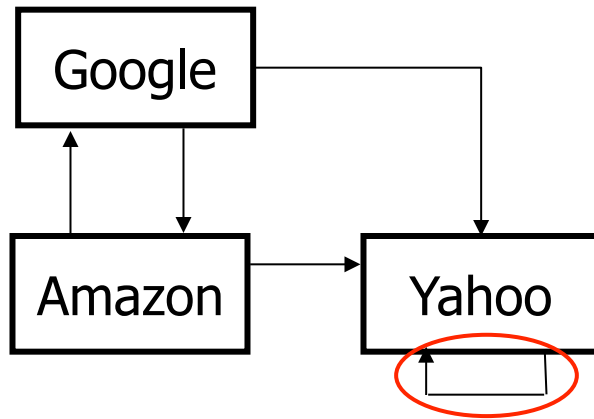
$$\begin{bmatrix} g' \\ y' \\ a' \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0.5 \\ 0.5 & 0 & 0.5 \\ 0.5 & 0 & 0 \end{bmatrix} * \begin{bmatrix} g \\ y \\ a \end{bmatrix}$$

'dead end' - PageRank is lost after each round

Running for multiple iterations:

$$\begin{bmatrix} g \\ y \\ a \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0.5 \\ 1 \\ 0.5 \end{bmatrix}, \begin{bmatrix} 0.25 \\ 0.5 \\ 0.25 \end{bmatrix}, \dots, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Oops #2 – PageRank hogs



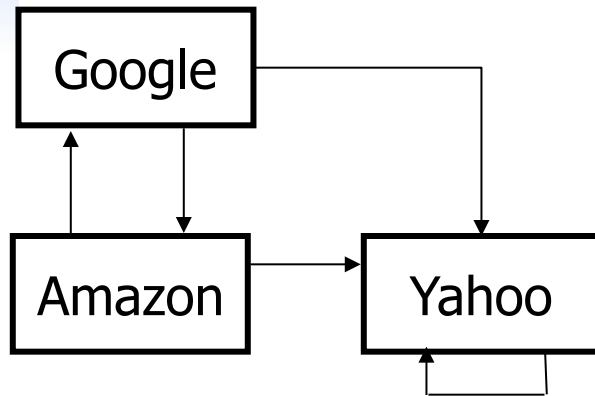
$$\begin{bmatrix} g' \\ y' \\ a' \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0.5 \\ 0.5 & 1 & 0.5 \\ 0.5 & 0 & 0 \end{bmatrix} * \begin{bmatrix} g \\ y \\ a \end{bmatrix}$$

PageRank cannot flow out and accumulates

Running for multiple iterations:

$$\begin{bmatrix} g \\ y \\ a \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0.5 \\ 2 \\ 0.5 \end{bmatrix}, \begin{bmatrix} 0.25 \\ 2.5 \\ 0.25 \end{bmatrix}, \dots, \begin{bmatrix} 0 \\ 3 \\ 0 \end{bmatrix}$$

Stopping the Hog



$$\begin{bmatrix} g' \\ y' \\ a' \end{bmatrix} = 0.85 \begin{bmatrix} 0 & 0 & 0.5 \\ 0.5 & 1 & 0.5 \\ 0.5 & 0 & 0 \end{bmatrix} * \begin{bmatrix} g \\ y \\ a \end{bmatrix} + \begin{bmatrix} 0.15 \\ 0.15 \\ 0.15 \end{bmatrix}$$

Running for multiple iterations:

$$\begin{bmatrix} g \\ y \\ a \end{bmatrix} = \begin{bmatrix} 0.57 \\ 1.85 \\ 0.57 \end{bmatrix}, \begin{bmatrix} 0.39 \\ 2.21 \\ 0.39 \end{bmatrix}, \begin{bmatrix} 0.32 \\ 2.36 \\ 0.32 \end{bmatrix}, \dots, \begin{bmatrix} 0.26 \\ 2.48 \\ 0.26 \end{bmatrix}$$

Improved PageRank

- Remove out-degree 0 nodes (or consider them to refer back to referrer)
- Add **decay factor d** to deal with sinks

$$PageRank(p) = (1 - d) + d \sum_{b \in B_p} \frac{1}{N(b)} PageRank(b)$$

- Typical value: $d=0.85$
- Intuition in the idea of the “**random surfer**”:
 - Surfer occasionally stops following link sequence and jumps to new random page, with probability $1 - d$

PageRank on MapReduce

■ Inputs

- Of the form: $\text{page} \rightarrow (\text{currentWeightOfPage}, \{\text{adjacency list}\})$

■ Map

- Page p “propagates” $1/N_p$ of its $d * \text{weight}(p)$ to the destinations of its out-edges (think like a vertex!)

■ Reduce

- p -th page sums the incoming weights and adds $(1-d)$, to get its $\text{weight}'(p)$

■ Iterate until convergence

- Common practice: run some fixed number of times, e.g., 25x
- Alternatively: Test after each iteration with a second MapReduce job, to determine the maximum change between old and new weights

Recap and Take-aways

- We've had a whirlwind tour of common kinds of algorithms used on the Web
 - Path analysis: route planning, games, keyword search, etc.
 - Clustering and classification: mining, recommendations, spam filtering, context-sensitive search, ad placement, etc.
 - Link analysis: ranking
- Many such algorithms (though not all) have a reasonably straightforward, often **iterative**, MapReduce formulation

Additional references

Data-Intensive Text Processing with MapReduce

Jimmy Lin and Chris Dyer

Morgan & Claypool Publishers, 2010

<http://lintool.github.io/MapReduceAlgorithms/>

Stay tuned



Next time you will learn about:
Beyond MapReduce