Lecture 5: MapReduce Applications

CSCI4180 (Fall 2013)

Patrick P. C. Lee

Acknowledgments

- Slides are adapted from Lectures 4-5 on http://www.umiacs.umd.edu/~jimmylin/cloud-2010-Spring/index.html

Recap

- Design patterns for MapReduce:
  - In-mapper combining
    - By preserving states in mappers
  - Pairs and stripes
    - Based on complex key/value types
  - Order inversion
    - Based on sorting property of intermediate key-value pairs
  - Value-to-key inversion
    - Providing the secondary sorting property
- Question: how to apply these design patterns into real-life applications?

Outline

- Text retrieval
  - Inverted indexing
- Graph algorithms
  - Parallel breadth-first search
  - Parallel Dijkstra's algorithm
  - PageRank

Web Search Problem

- Web search is to retrieve relevant web objects
  - e.g., web pages, PDFs, PPT slides
- Web search problem:
  - Crawling: gathering web content
  - Indexing: constructing search indexing structure
  - Retrieval: ranking documents given a query

Web Search Problem

- Challenge:
  - The web is huge
  - Billions of web objects, terabytes of information
- Performance goals:
  - Query latency needs to be small
    - Within hundred of milliseconds
  - Scalable for a large number of documents

Web Crawling

- Features of real-life web crawlers:
  - Not overload web servers
  - Prioritize the pages to be fetched
  - Deployed as a distributed system
  - Adapt to the frequency of web content changes
  - Identify web page duplicates
  - Multi-lingual support

Inverted Indexes

用來儲存在全文搜索下單字(word)與單字本身在文件中所在位置之間的映射

- **Inverted index** 找到某個單字在文件檔中出現的地方
  - A data structure that given a term provides access to the list of documents that contain the term
  - Used by most full-text search engines today
  - By documents, we mean web objects
- Retrieval engine uses the inverted index to score documents that contain the query terms based on some ranking model
  - e.g., based on term matches, term proximity, term attributes, etc.

Inverted Indexes

- Illustration:

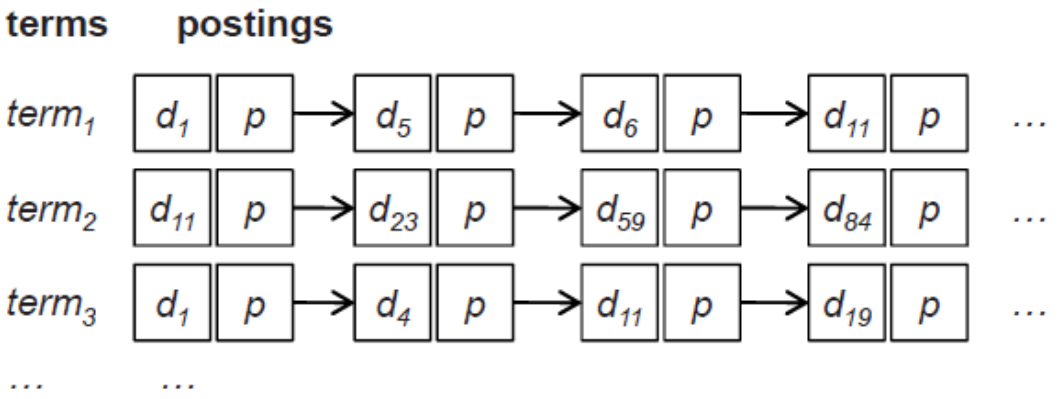


Figure 4.1: Simple illustration of an inverted index. Each term is associated with a list of postings. Each posting is comprised of a document id and a payload, denoted by *p* in this case. An inverted index provides quick access to documents ids that contain a term.

Inverted Indexes

- **Posting**: a tuple of (document id, payload) associated with a term
- Document ID:
  - Assumed to be a unique integer from 1 to n, where n is the total number of documents
- Payload:
  - Could be empty for simple Boolean retrieval
  - Could include term frequency (tf), the number of times the term occur in the document

Inverted Indexes

- Given a query, retrieval involves fetching postings lists associated with query terms and traversing the postings to compute the result set.
- Simple Boolean retrieval:
  - Apply union (OR) or intersection (AND) of posting lists
- General retrieval:
  - Document scores are ranked
  - Top k documents are returned

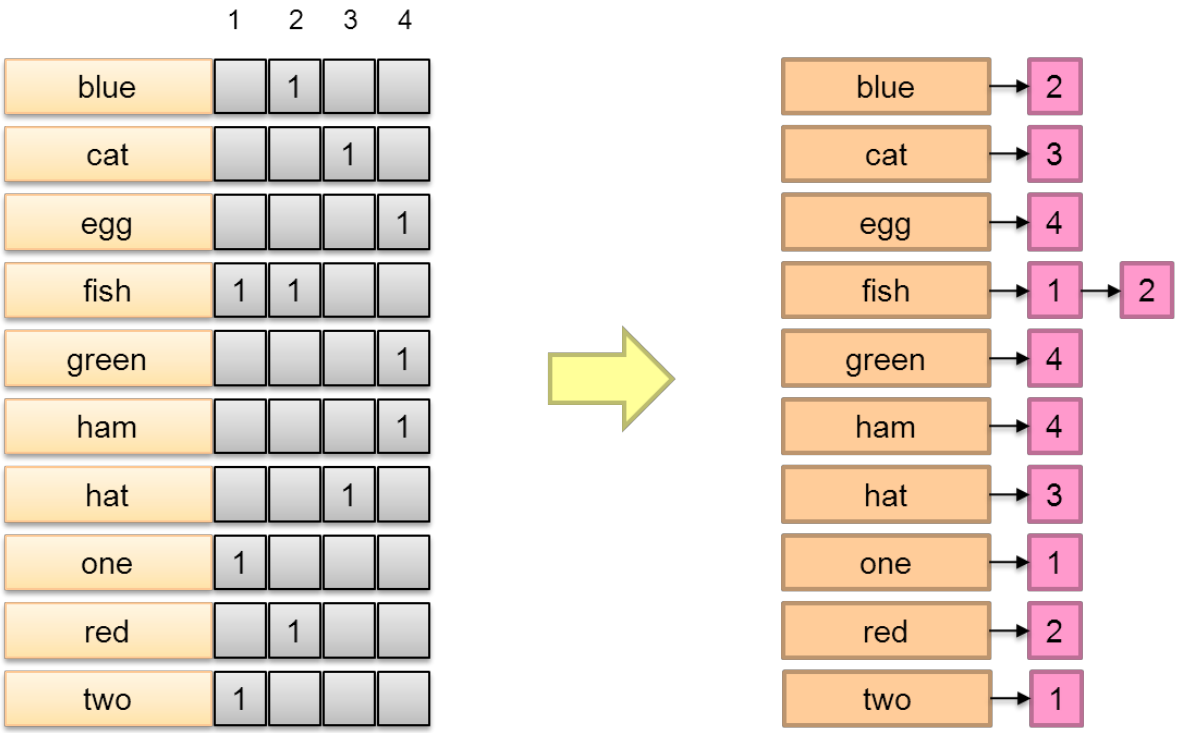
Inverted Indexes: Boolean Retrieval

Doc 1 one fish, two fish

Doc 2 red fish, blue fish

Doc 3 cat in the hat

Doc 4 green eggs and ham



Inverted Indexes: Construction

- **How to construct an inverted index?** 
- Naive approach:
  - For each document, extract all useful terms, and exclude all stopwords (e.g., ‘the’, ‘a’, ‘of’) and remove affixes (e.g., ‘dogs’ to ‘dog’)
  - For each term, add the posting (document, payload) to an existing list, or create a posting list if the term is new
- Clearly, naive approach is not scalable if the document collection is huge and each document is large
- Can we use MapReduce?

Baseline Implementation

- Our goal: construct an inverted index given a document collection
- Main idea:
  - Input to each mapper:
    - Document IDs (keys)
    - Actual document content (values)
  - What each mapper does:
    - Analyze each document and extract useful terms
    - Compute term frequencies (per document)
    - Emit (term, posting)
  - What each reducer does
    - Aggregates all observed postings for each term
    - Construct the posting list

Baseline Implementation

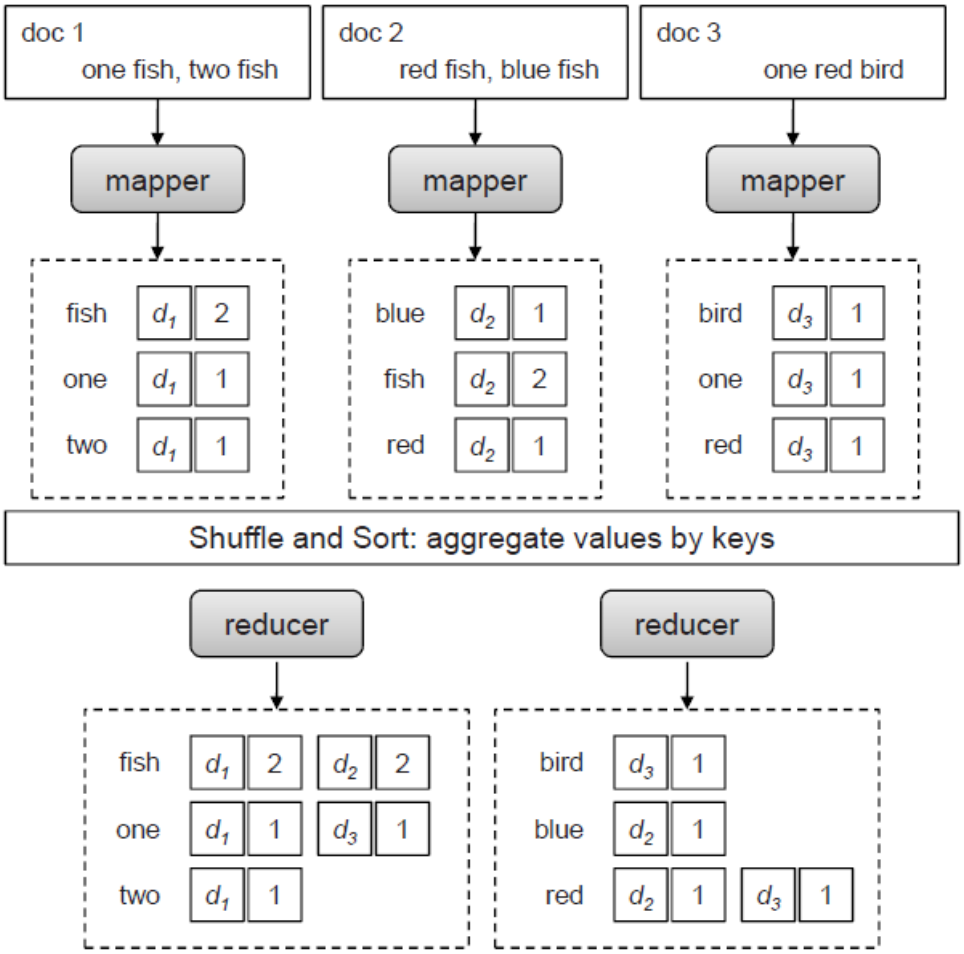
- Pseudo-code:

```
1: class Mapper
2:     procedure Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)

1: class Reducer
2:     procedure Reduce(term t, postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .])
3:         P ← new List
4:         for all posting ⟨a, f⟩ ∈ postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .] do
5:             Append(P, ⟨a, f⟩)
6:         Sort(P)        Need to sort explicitly in reducer
7:         Emit(term t, postings P)
```

Baseline Implementation

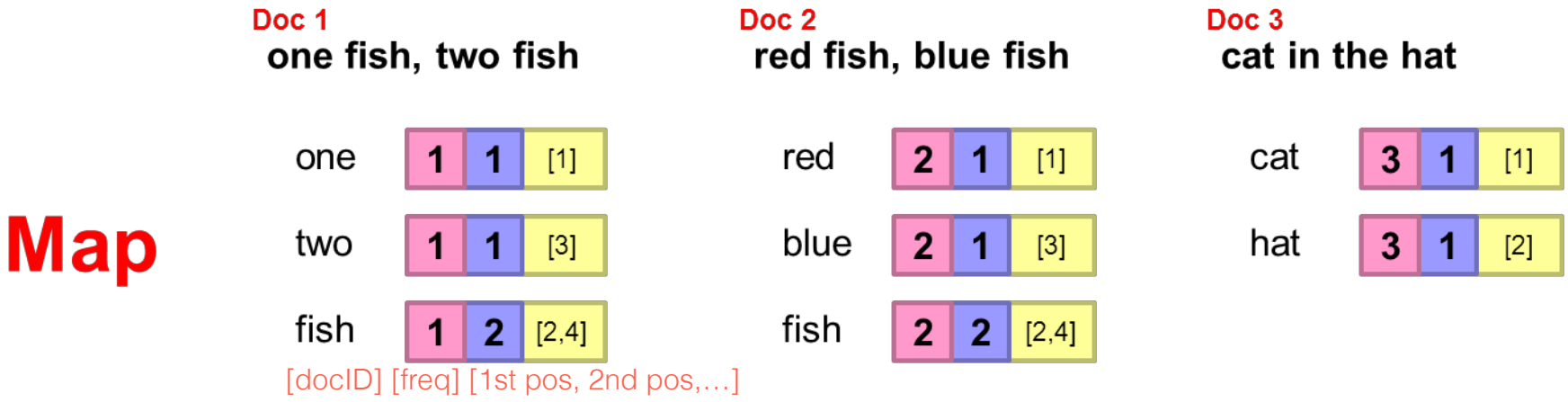
- Illustration:



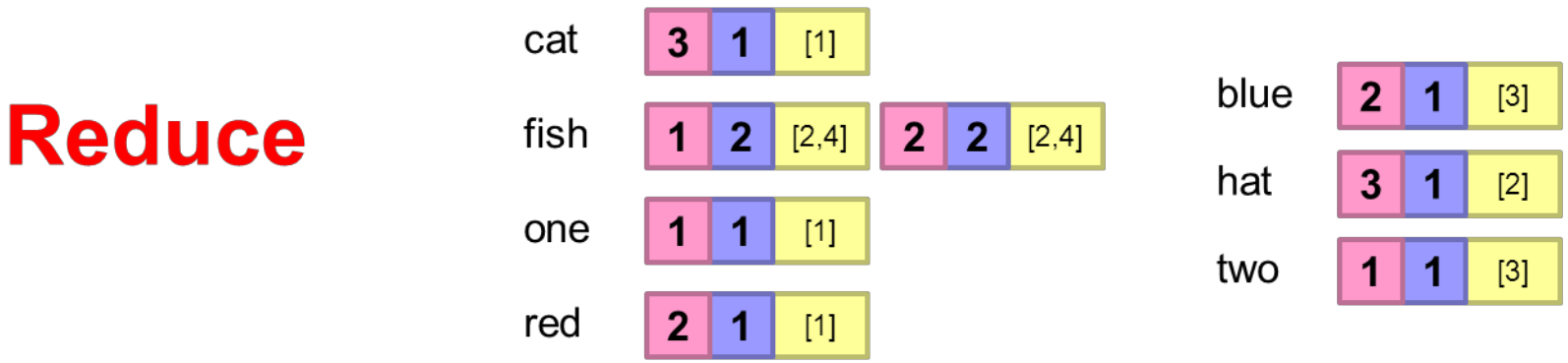
Baseline Implementation

- In the shuffle and sort phase, MapReduce framework forms a large, distributed group by the postings of each term
- From reducer's point of view
  - Each input to the reducer is the resulting posting list of a term
  - Reducer may sort the list (if needed), and writes the final output to disk
  - The task of each reducer is greatly simplified! MapReduce framework has done most heavy liftings.

Positional Indexes



Shuffle and Sort: aggregate values by keys



Scalability Issue

- Scalability problem in baseline implementation

```
1: class Mapper
2:     procedure Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)

1: class Reducer
2:     procedure Reduce(term t, postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .])
3:         P ← new List
4:         for all posting ⟨a, f⟩ ∈ postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .] do
5:             Append(P, ⟨a, f⟩)
6:         Sort(P)
7:         Emit(term t, postings P)
```

Any problem?

Scalability Issue

- Assumption of baseline implementation:
  - Reducer has sufficient memory to hold all postings associated with the same term
- Why?
  - The MapReduce framework makes no guarantees about the ordering of values associated with the same key.
  - The reducer first buffers all postings (line 5) and then performs an in-memory sort before writing the postings to disk

Scalability Issue

- How to solve? Key idea is to let MapReduce framework do sorting for us

make use of the hdfs space to keep data

- Instead of emitting

  (term t, posting <docid, f>)

  Emit

  (tuple <t, docid>, f)

- Value-to-key conversion!!

Revised Implementation

- With value-to-key conversion, the MapReduce framework ensures the postings arrive in sorted order (based on <term t, docid>)
- Results can be written to disk directly

- Caution: you need a customized partitioner to ensure that all tuples with the same term are shuffled to the same reducer

Revised Implementation

```
1: class Mapper
2:     method Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(tuple ⟨t, n⟩, tf H{t})

1: class Reducer
2:     method Initialize
3:         t_prev ← ∅
4:         P ← new PostingsList
5:     method Reduce(tuple ⟨t, n⟩, tf [f])
6:         if t ≠ t_prev ∧ t_prev ≠ ∅ then
7:             Emit(term t, postings P)
8:             P.Reset()
9:         P.Add(⟨n, f⟩)
10:        t_prev ← t
11:    method Close
12:        Emit(term t, postings P)
```


Results are directly written to disk

Retrieval

- Let's revisit the web search problem:
  - Crawling: brief discussion about design features
  - Indexing:
    - Fundamentally a batch operation
    - Need to be fast, but need not be in real time
    - Scalability is an issue
  - Retrieval:
    - Response time needs to be very small

- Should we apply MapReduce to retrieval?

Retrieval

- MapReduce is designed for large batch operations, and is a poor solution for retrieval
  - MapReduce is optimized for throughput, but not for latency
- Retrieval in HDFS is also time-consuming
  - What does retrieval do?
    - Look up posting lists corresponding to query terms
    - Traverse posting lists to compute scores
    - Return top k results
  - Many random seeks will be introduced; many round-trips in distributed file systems

Retrieval

- How to make retrieval fast?
- Use **distributed retrieval**, and partition indexes across multiple servers
- Partitioning strategies:
  - Document partitioning
    - Entire collection is partitioned into smaller sub-collections
  - Term partitioning
    - Each server handles a subset of terms for the entire collection

Retrieval

- Document partitioning is generally faster:
  - Parallelize the search of a query term over multiple servers, each handling a subset of documents
- Term partitioning aggregates the search of a query term in one single server
  - Popular terms may create “hot spots” on servers
  - But it generally involves fewer disk seeks since the results are aggregated for each term

Outline

- Text retrieval
  - Inverted indexing
- Graph algorithms
  - Parallel breadth-first search
  - Parallel Dijkstra's algorithm
  - PageRank

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

Real-World Graph Problems

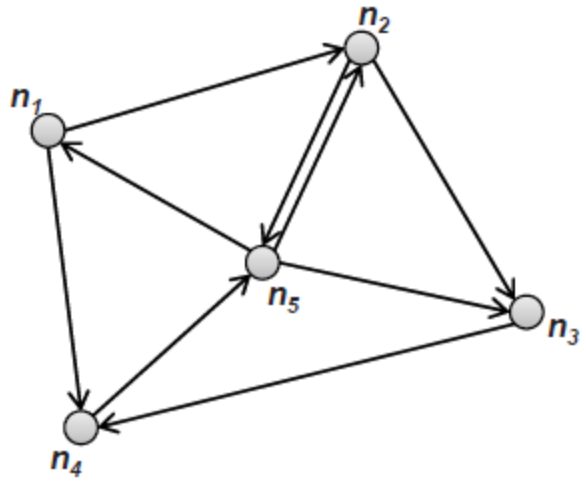
- Finding shortest paths
  - Routing Internet traffic and UPS trucks
- Finding minimum spanning trees 從一張圖上分離出一棵包含圖上所有點的樹
  - Telco laying down fiber
- Finding Max Flow
  - Airline scheduling
- Identify “special” nodes and communities
  - Breaking up terrorist cells, spread of avian flu
- Bipartite matching
  - Monster.com, Match.com
- PageRank

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
- Challenge:
  - Algorithms running on a single machine and putting the entire graph in memory are not scalable
- Key questions:
  - How do you represent graph data in MapReduce?
  - How do you traverse a graph in MapReduce?

Graph Representations

- Two common representations
  - Adjacency matrix
  - Adjacency list



	n_1	n_2	n_3	n_4	n_5
n_1	0	1	0	1	0
n_2	0	0	1	0	1
n_3	0	0	0	1	0
n_4	0	0	0	0	1
n_5	1	1	1	0	0

adjacency matrix

n_1	$[n_2, n_4]$
n_2	$[n_3, n_5]$
n_3	$[n_4]$
n_4	$[n_5]$
n_5	$[n_1, n_2, n_3]$

adjacency lists

Adjacency Matrix

- Advantages:
  - Easy to manipulate with linear algebra
  - Easy algorithmic implementation
- Disadvantage:
  - Large memory space, especially for sparse graphs (Many nodes but few edges)
    - Sparse graphs: number of actual edges is far smaller than the number of possible edges
    - Many zeroes in the matrix

Adjacency Lists

Good for map reduce framework

- Advantages:
  - Much more compact representation
  - Easy to compute over out-links
- Disadvantages:
  - Much more difficult to compute over in-links

- However, the shuffle and sort mechanism in MapReduce provides an easy way to group edges by destination nodes, thus allowing us to compute over incoming edges in the reducer

Single-Source Shortest Path

- **Problem**: find shortest paths from a source node to all other nodes in the graph
  - Shortest mean smallest hop counts or lowest weights
- Algorithm: Assume no negative cycle
  - Breadth-first-search: for finding minimum hop counts
  - Dijkstra's algorithm: for finding minimum-cost paths for general graphs

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)
```

Figure 5.2: Pseudo-code for Dijkstra's algorithm, which is based on maintaining a global priority queue of nodes with priorities equal to their distances from the source node. At each iteration, the algorithm expands the node with the shortest distance and updates distances to all reachable nodes.

Dijkstra's Algorithm

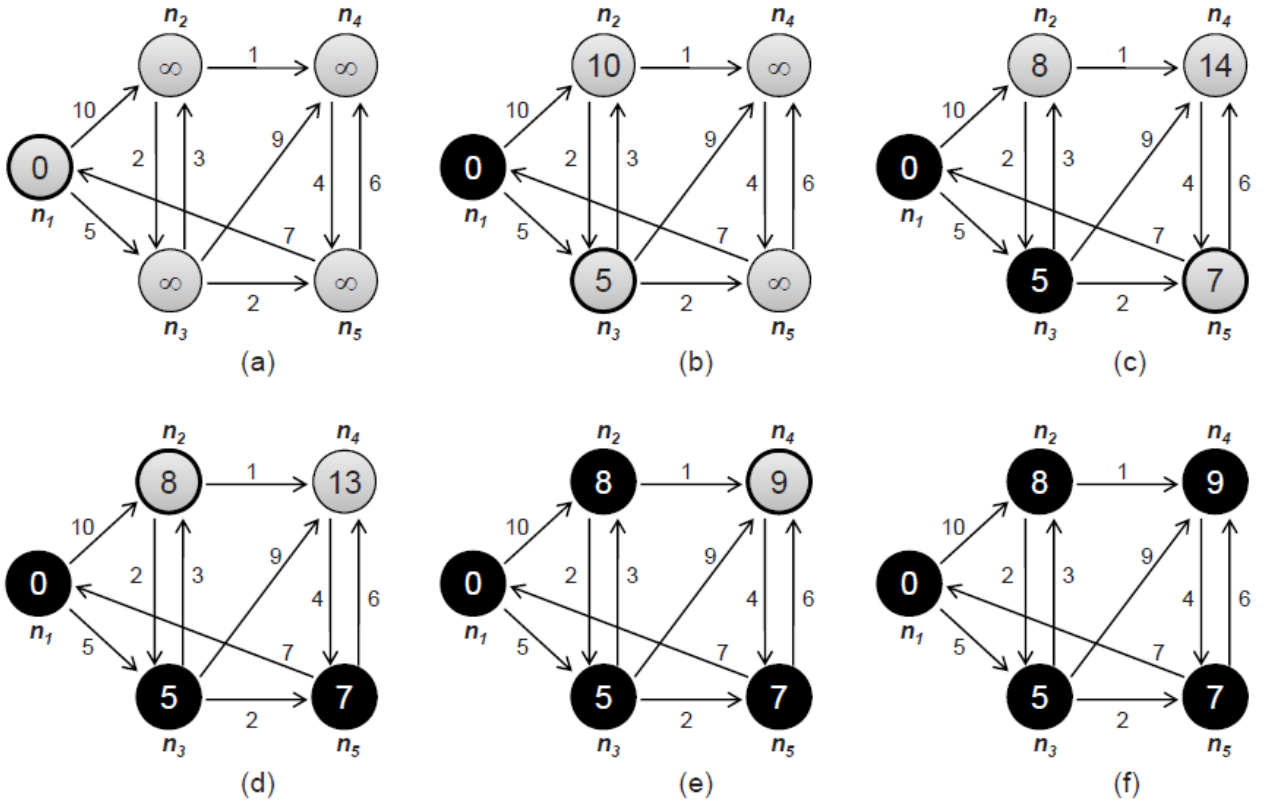


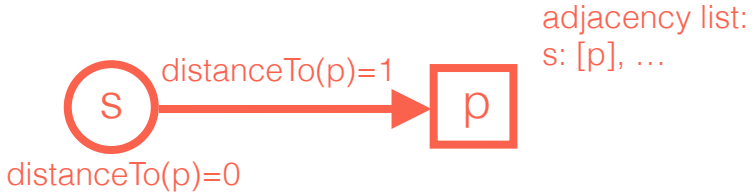
Figure 5.3: Example of Dijkstra's algorithm applied to a simple graph with five nodes, with n_1 as the source and edge distances as indicated. Parts (a)–(e) show the running of the algorithm at each iteration, with the current distance inside the node. Nodes with thicker borders are those being expanded; nodes that have already been expanded are shown in black.

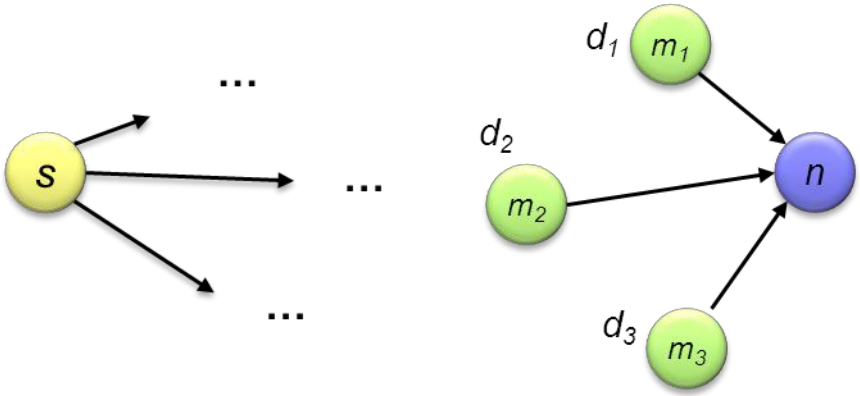
Dijkstra's Algorithm

- Dijkstra's algorithm is designed as a sequential algorithm
- Key to Dijkstra's algorithm
  - Priority queue that maintains a globally sorted list of nodes by current distance
  - Not possible in MapReduce, which doesn't provide a mechanism for exchanging global data
- Solution:
  - Brute-force approach: parallel breadth first search
    - Brute force: Try to revisit many nodes that have been visited

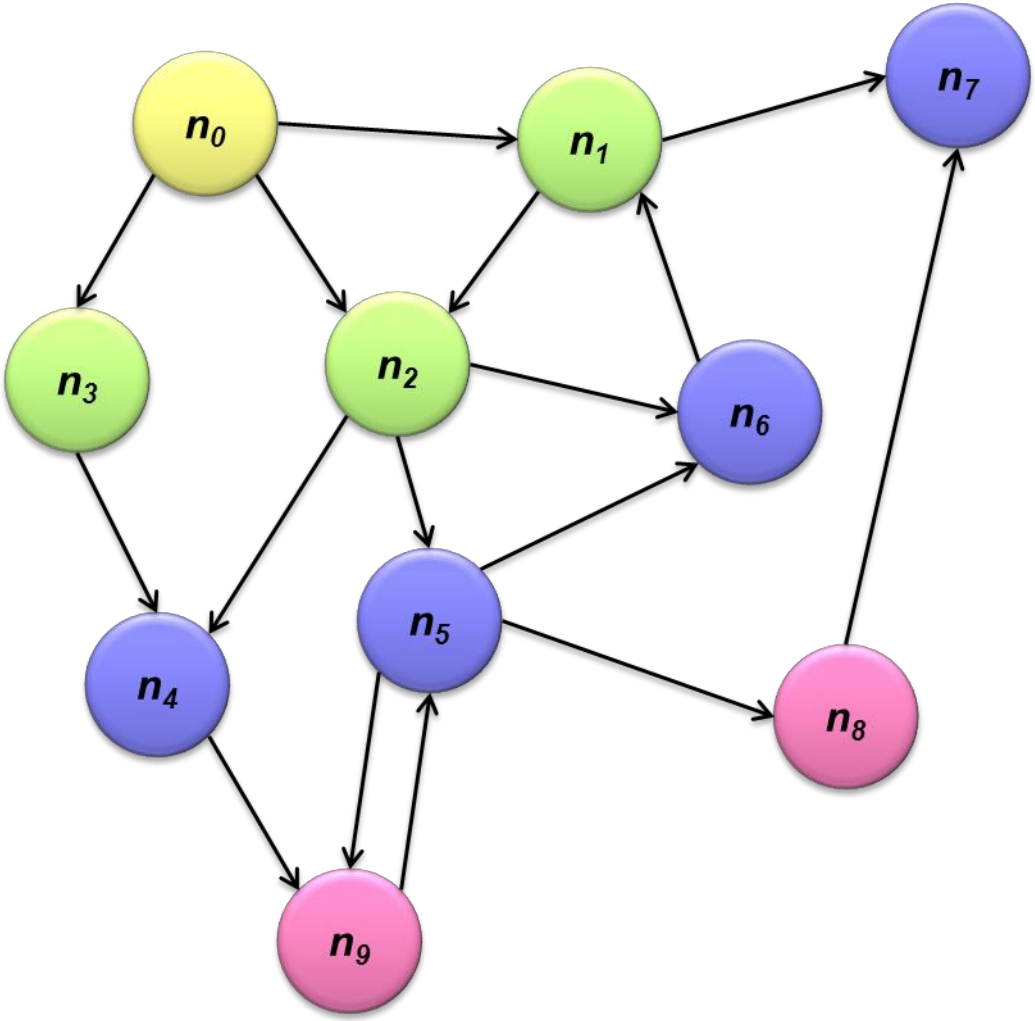
Parallel BFS

- 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*
  - 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* ∈ *M*)





Parallel BFS: Visualization



Intuition

- 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

- Multiple MapReduce iterations are needed
  - Each iteration corresponds to a MapReduce job
- Each MapReduce iteration advances the "known 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*, adjacency list) as well

Parallel BFS: Pseudo-code

```
1: class Mapper
2:     method Map(nid n, node N)
3:         d ← 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 ← ∞
4:         M ← ∅
5:         for all d ∈ counts [d1, d2, ...] do
6:             if IsNode(d) then
7:                 M ← d                           ▷ Recover graph structure
8:             else if d < d_min then              ▷ Look for shorter distance
9:                 d_min ← d
10:        M.Distance ← d_min                      ▷ Update shortest distance
11:        Emit(nid m, node M)
```

Figure 5.4: Pseudo-code for parallel breath-first search in MapReduce: the mappers emit distances to reachable nodes, while the reducers select the minimum of those distances for each destination node. Each iteration (one MapReduce job) of the algorithm expands the "search frontier" by one hop.

Stopping Criterion

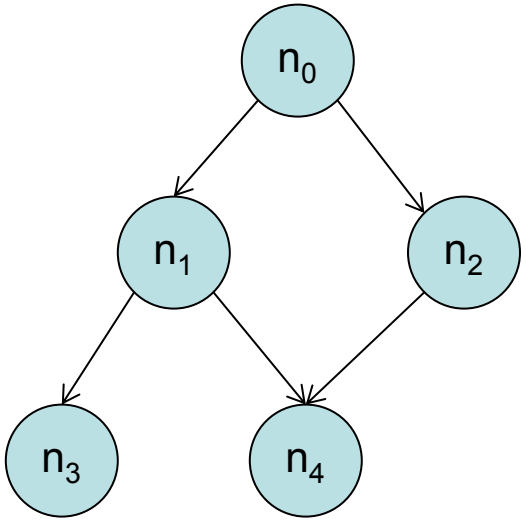
- How many iterations are needed in parallel BFS (equal edge weight case)?
- Convince yourself: when a node is first "discovered", we've found the shortest path
- Answer:
  - The diameter of the graph (i.e., greatest distance between any pair of node)
  - Six degrees of separation?

Stopping Criterion

- In practice, we iterate the algorithm until all node distances are found (i.e., no more ∞)
- How?
  - Maintain a counter inside the MapReduce program (i.e., count how many node distances are found)
  - Require a non-MapReduce driver program to submit a MapReduce job to iterate the algorithm
  - The driver program checks the counter value before submitting another job

In-Class Exercise

- How to find the shortest paths from n_0 to all other nodes?



Extend to General Weights

- Up until now, we have been assuming that all edges are unit distance.
- Simple change: adjacency list now includes a weight *w* for each edge
  - In mapper, emit (*m*, *d* + *w*) instead of (*m*, *d* + 1) for each node *m*
  - Note that w must be positive. (Why?)
- No other changes are required, but the termination behavior is very different

Stopping Criterion

- How many iterations are needed in parallel BFS (positive edge weight case)?
- In the worst case, we might need as many iterations as number of nodes in the graph minus one
  - This worst case is achievable
- How do we know that all shortest path distances are found?

Key Points of Graph Algorithms

- 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”
- It’s important to pass the graph structure between iterations

Random Walks over the Web

- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: "traversing" the graph
- Random surfer model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- Intuitively, under the random surfer model a page that is clicked more will be more popular

PageRank

- PageRank
  - A measure of web page quality
  - Used by Google's search algorithm
  - Characterizes the amount of time spent on any given page, or how frequently a page is encountered by a random surfer
  - 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
  - Note: query-independent

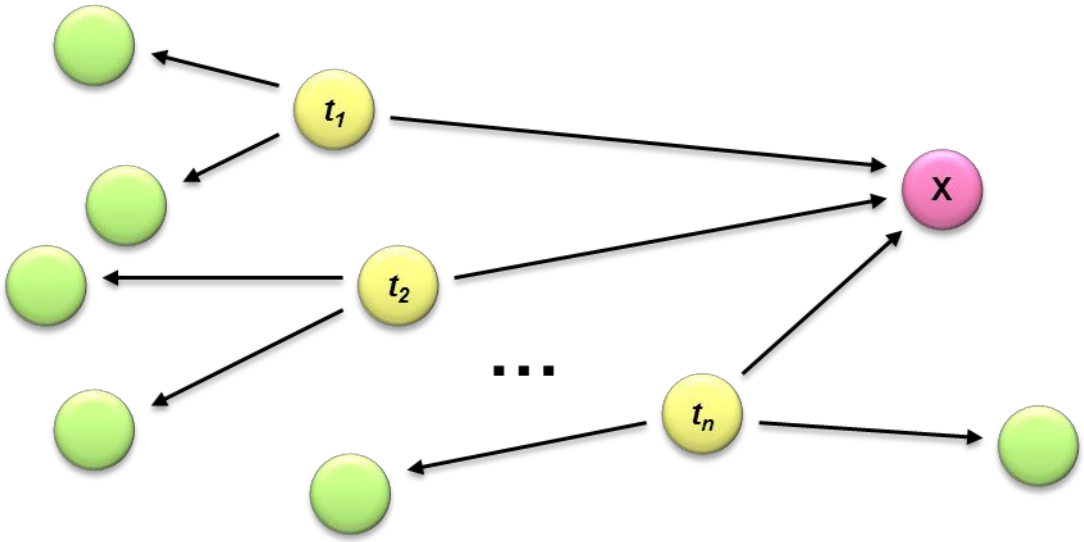
Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the Web. Stanford Digital Library Working Paper SIDL-WP-1999-0120, Stanford University, 1999

PageRank

- Given page *x* with inlinks $t_1 \ldots t_n$, where
  - *C(t)* is the out-degree of *t*
  - α is probability of random jump
  - *N* is the total number of nodes in the graph

Type the actual location in browser

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



PageRank

- Is PageRank robust?
  - Can anyone "game" PageRank?
- Spider trap
  - An infinite chain of pages (e.g., generated by CGI) that all link to a single page, so as to inflate the PageRank value
- But let's assume a community of honest users

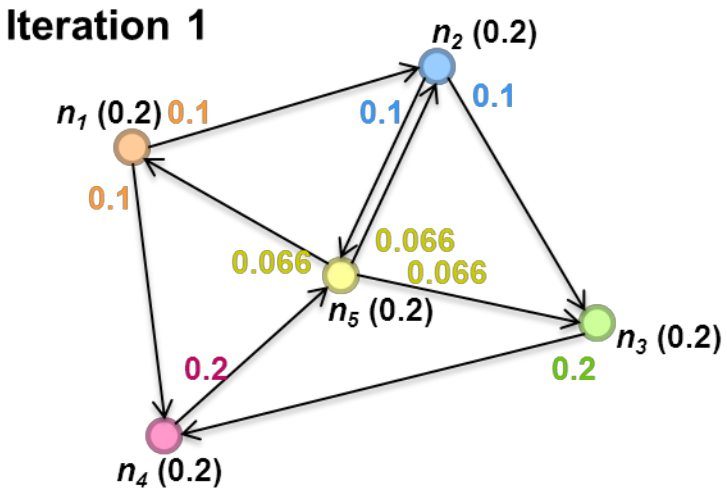
Simplified PageRank

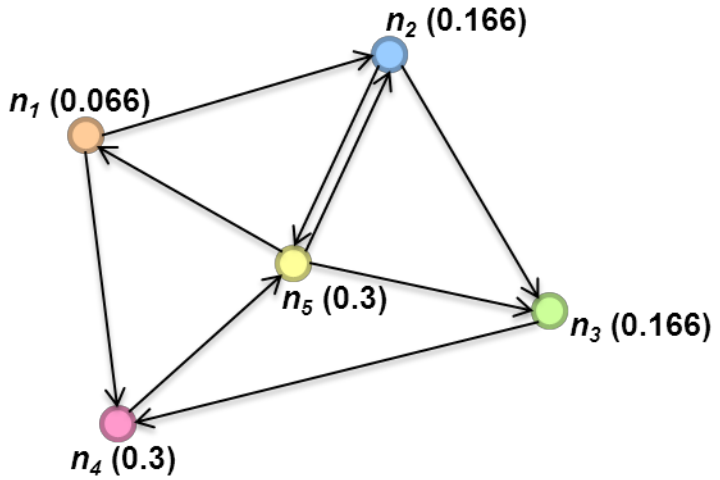
- First, tackle the simple case:
  - No random jump factor
    - i.e., $\alpha = 0$
  - No dangling links
    - i.e., every page must have at least one outgoing link

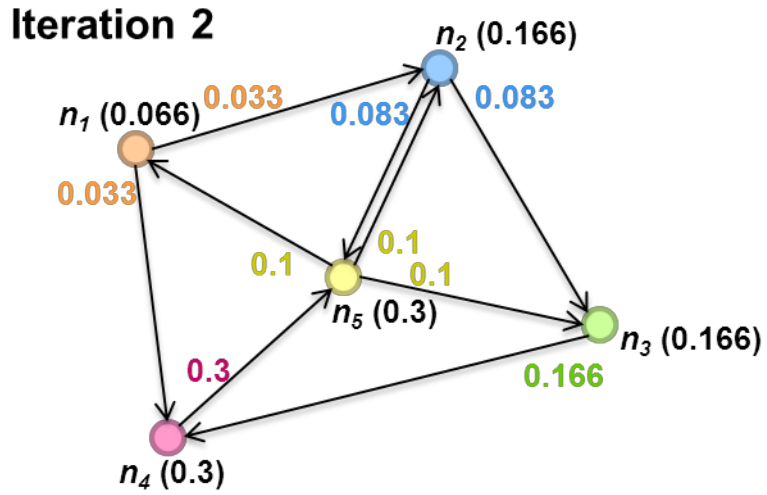
Computing PageRank

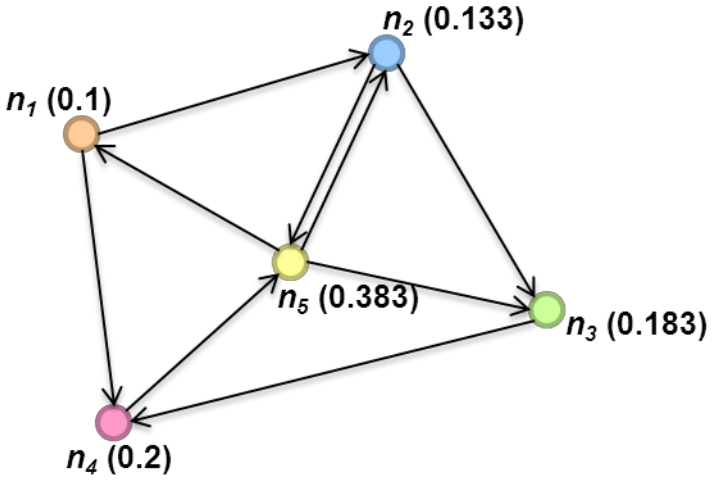
- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm:
  - Start with seed PR_i values (e.g., every page has equal PageRank)
  - Each page distributes PR_i "credit" to all pages it links to
  - Each target page adds up "credit" from multiple in-bound links to compute PR_{i+1}
  - Iterate until values converge

Toy Example


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


n_2 (0.166)
n_1 (0.066)
n_5 (0.3)
n_3 (0.166)
n_4 (0.3)


Iteration 2
n_2 (0.166)
n_1 (0.066)
0.033
0.083
0.083
0.033
0.1
0.1
0.1
n_5 (0.3)
n_3 (0.166)
0.3
0.166
n_4 (0.3)


n_2 (0.133)
n_1 (0.1)
n_5 (0.383)
n_3 (0.183)
n_4 (0.2)

PageRank in MapReduce

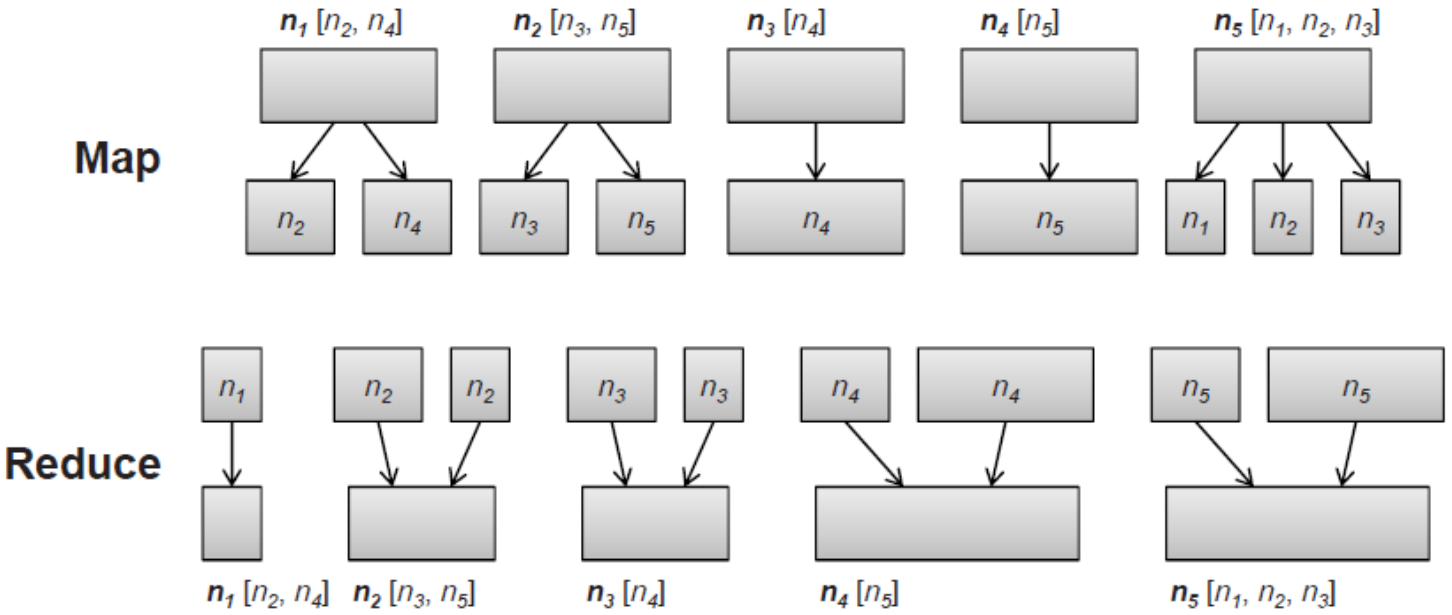


Figure 5.9: Illustration of the MapReduce PageRank algorithm corresponding to the first iteration in Figure 5.7. The size of each box is proportion to its PageRank value. During the map phase, PageRank mass is distributed evenly to nodes on each node's adjacency list (shown at the very top). Intermediate values are keyed by node (shown inside the boxes). In the reduce phase, all partial PageRank contributions are summed together to arrive at updated values.

PageRank in MapReduce

```
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                       ▷ Sum incoming PageRank contributions
9:         M.PageRank ← s
10:        Emit(nid m, node M)
```

Figure 5.8: Pseudo-code for PageRank in MapReduce (leaving aside dangling nodes and the random jump factor). In the map phase we evenly divide up each node's PageRank mass and pass each piece along outgoing edges to neighbors. In the reduce phase PageRank contributions are summed up at each destination node. Each MapReduce job corresponds to one iteration of the algorithm.

PageRank in MapReduce

- Map phase:
  - For each node, computes how much PageRank mass is emitted as value
- Shuffle and sort phase:
  - Group values passed along the graph edges by destination nodes
- Reduce phase:
  - PageRank mass contributions from all incoming edges are summed to arrive at the updated PageRank value for each node

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

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

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

PageRank Convergence

- Alternative convergence criteria
  - Iterate until PageRank values don't change
  - Iterate until PageRank rankings don't change
  - Fixed number of iterations
- The original PageRank paper shows that convergence on a graph with 322 million edges was reached in 52 iterations
- Yet, pages are dynamic, and PageRank algorithm need to be run from time to time