Data-Intensive Information Processing Applications — Session #5

Graph Algorithms

Jordan Boyd-Graber
University of Maryland

Thursday, March 3, 2011

Old Business

- HW1 Graded
  - Combiners throw away data!
- HW2 Due
- Last week slides updated
- Dense Representations
- Dumbo

Source: Wikipedia (Japanese rock garden)

Today's Agenda

- Graph problems and representations
- Parallel breadth-first search
- PageRank

What's a graph?

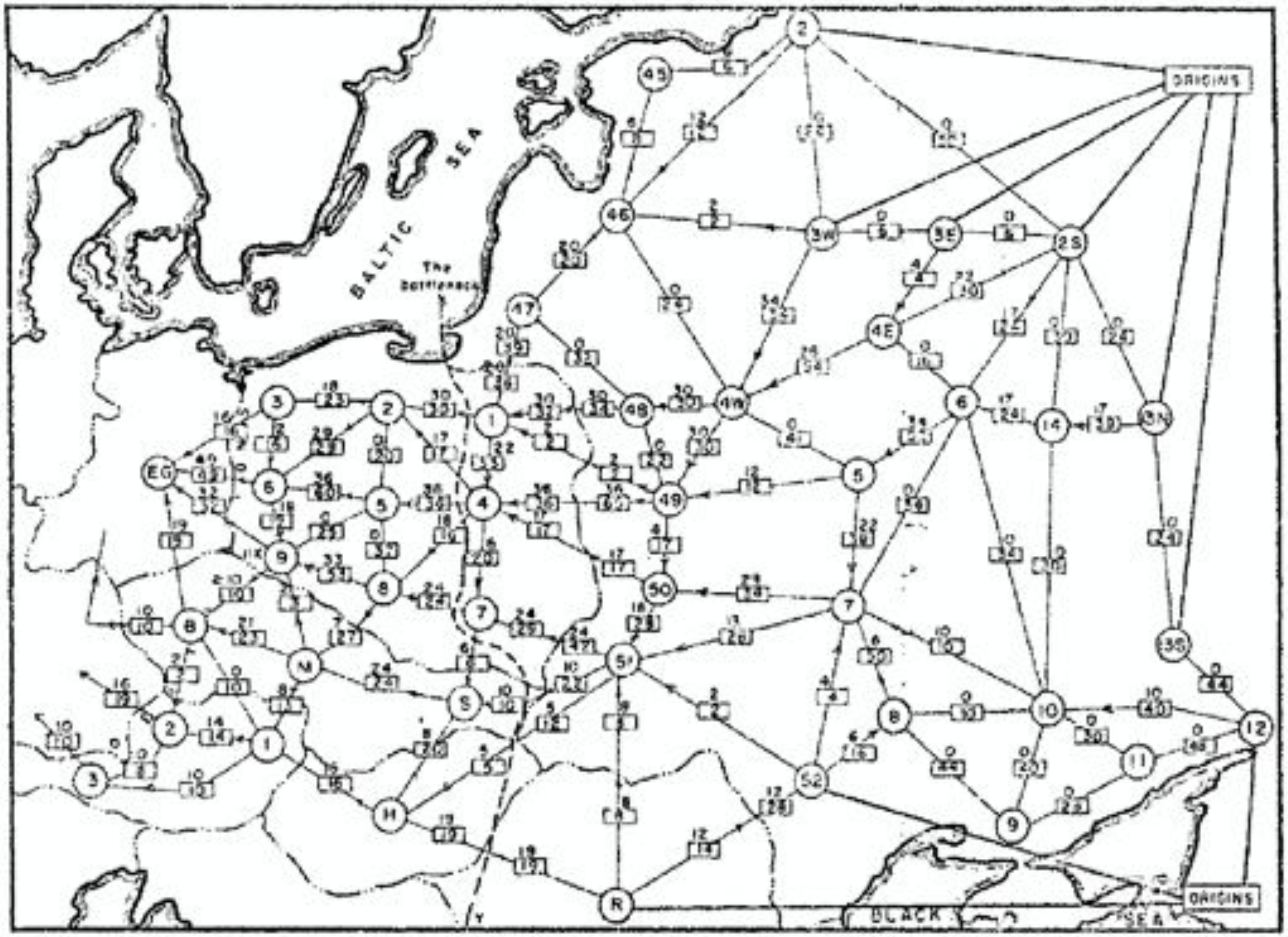
- G = (V,E), where
  - V represents the set of vertices (nodes)
  - E represents the set of edges (links)
  - Both vertices and edges may contain additional information
- Different types of graphs:
  - Directed vs. undirected edges
  - Presence or absence of cycles
- Graphs are everywhere:
  - Hyperlink structure of the Web
  - Physical structure of computers on the Internet
  - Interstate highway system
  - Social networks

Source: Wikipedia (Königsberg)

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

Max Flow / Min Cut



Reference: On the history of the transportation and maximum flow problems.
Alexander Schrijver in Math Programming, 91: 3, 2002.

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
- Key questions:
  - How do you represent graph data in MapReduce?
  - How do you traverse a graph in MapReduce?

Representing Graphs

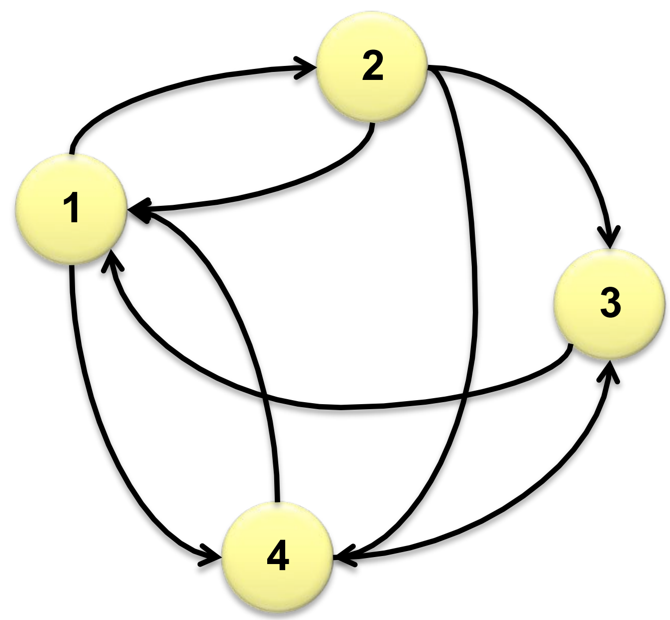
- G = (V, E)
- Two common representations
  - Adjacency matrix
  - Adjacency list

Adjacency Matrices

Represent a graph as an *n* x *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

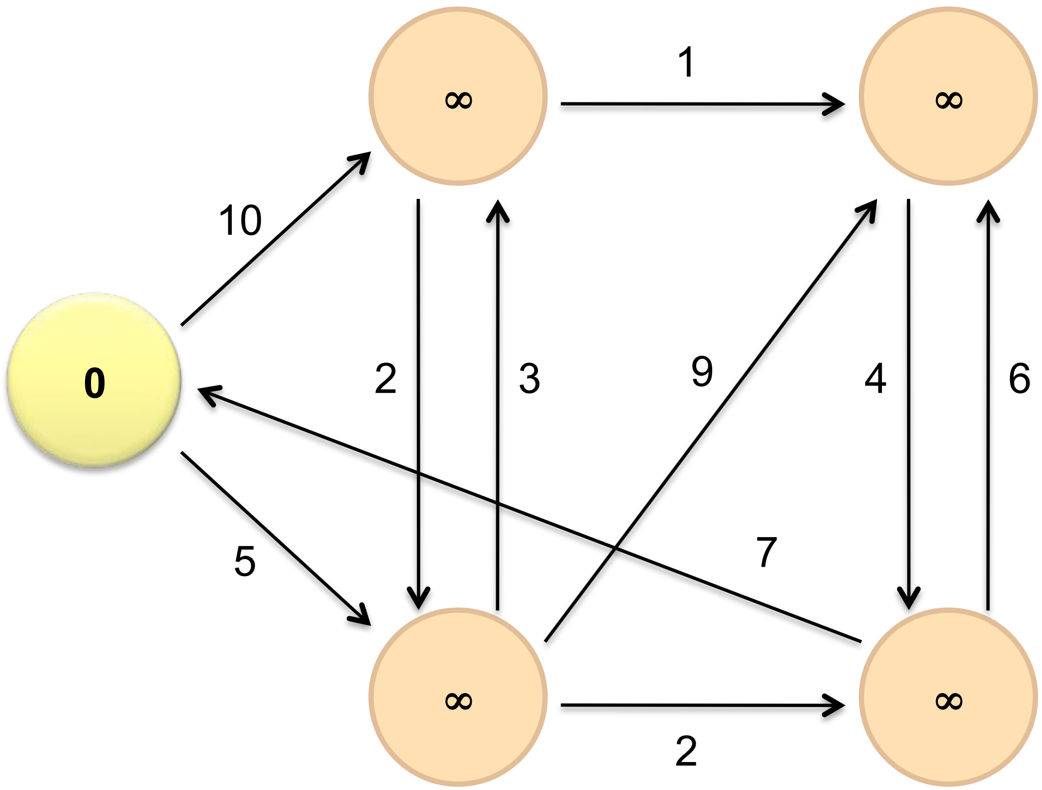
Adjacency Lists: Critique

- Advantages:
  - Much more compact representation
  - Easy to compute over outlinks
- Disadvantages:
  - Much more difficult to compute over inlinks

Single Source Shortest Path

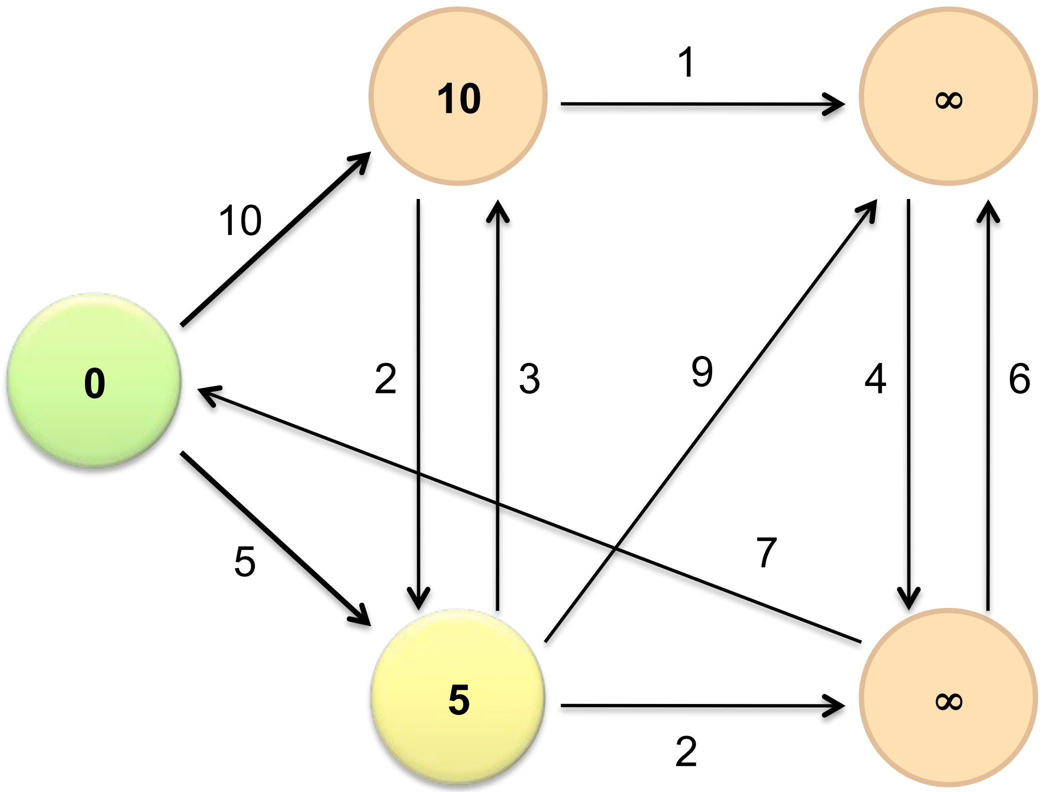
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- First, a refresher: Dijkstra's Algorithm

Dijkstra's Algorithm Example



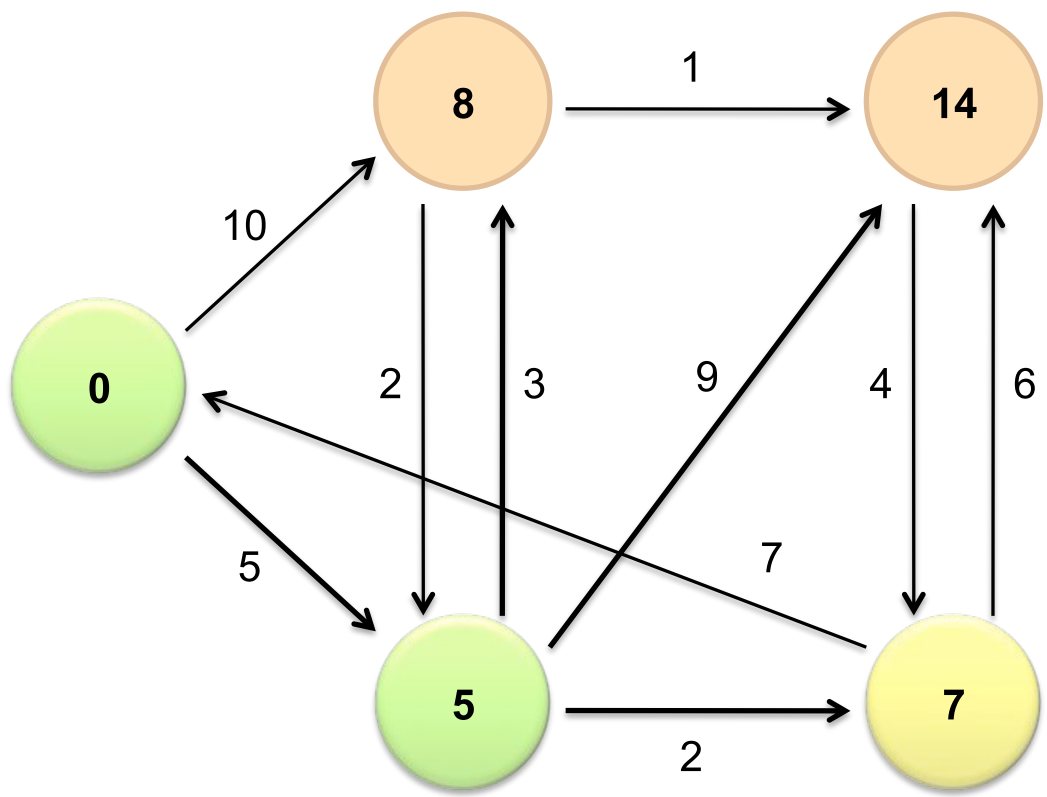
Example from CLR

Dijkstra's Algorithm Example



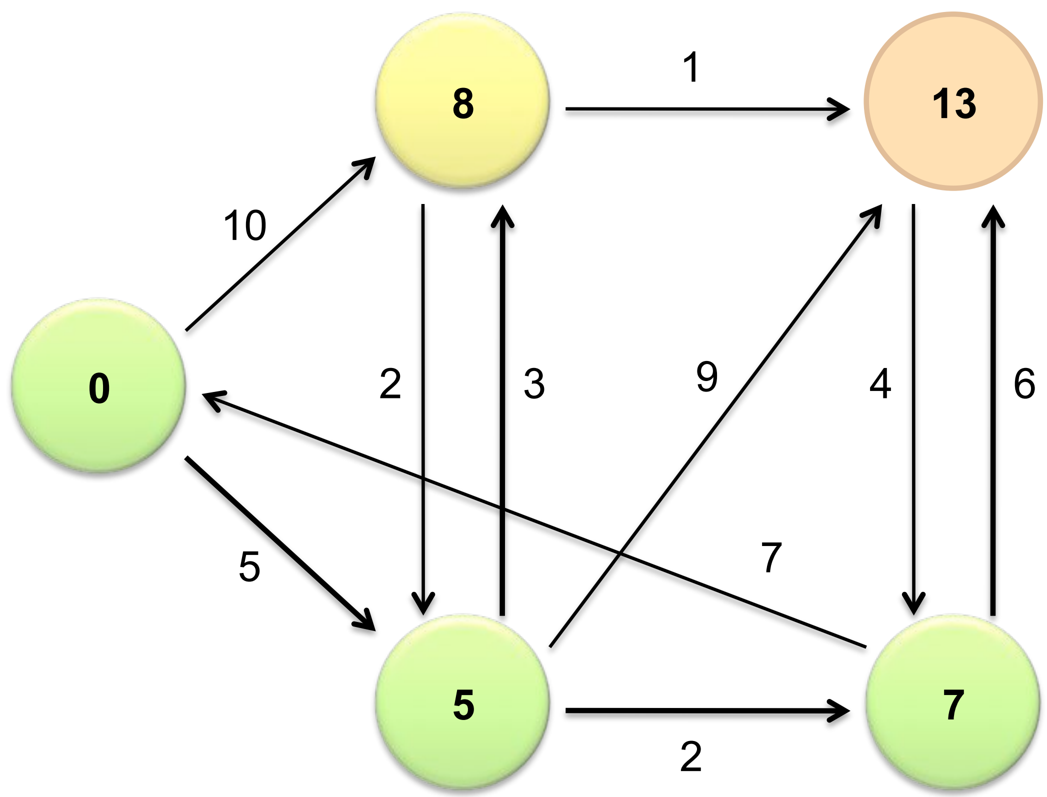
Example from CLR

Dijkstra's Algorithm Example



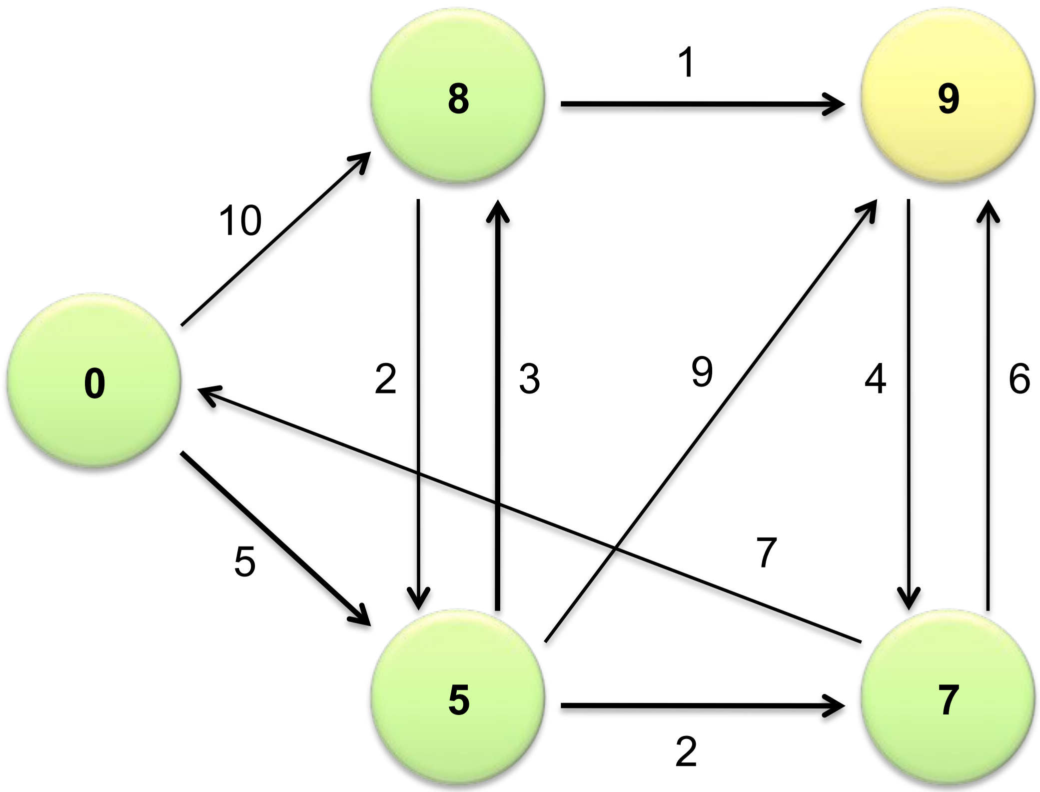
Example from CLR

Dijkstra's Algorithm Example



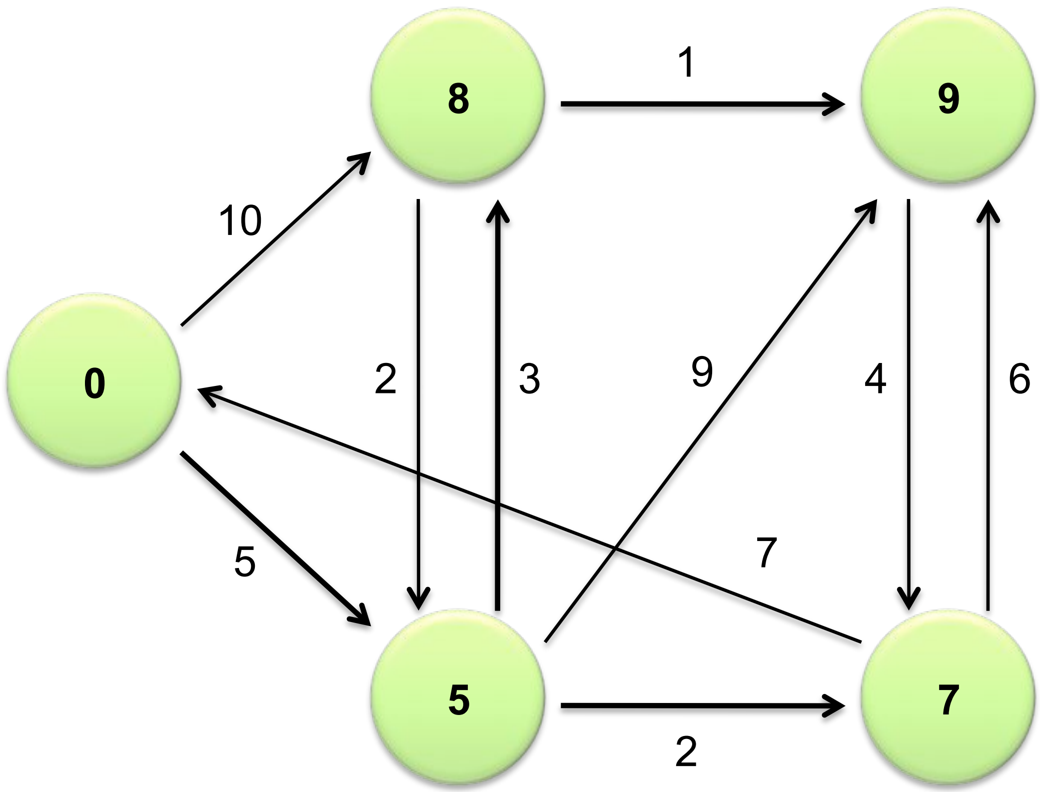
Example from CLR

Dijkstra’s Algorithm Example



Example from CLR

Dijkstra's Algorithm Example



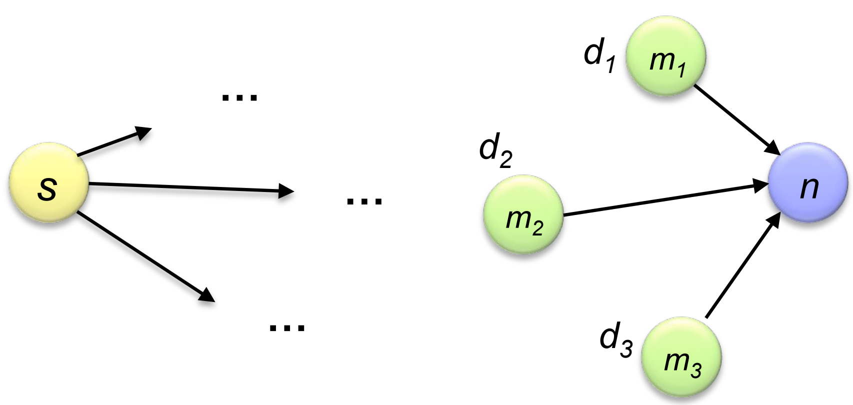
Example from CLR

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)

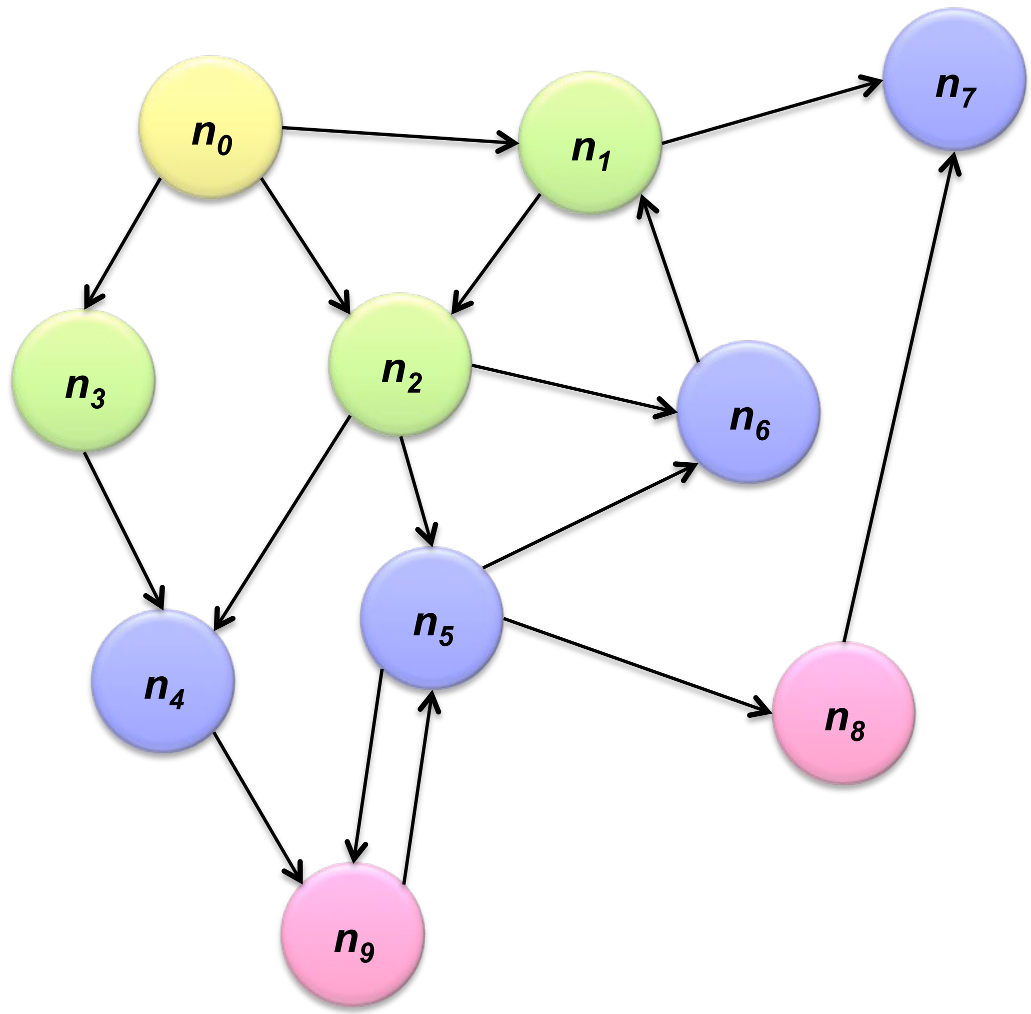
Finding the Shortest Path

- 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), $m \in M$)



Source: Wikipedia (Wave)

Visualizing Parallel BFS



From Intuition to Algorithm

- 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 Needed

- 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

BFS Pseudo-Code

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

class REDUCER
    method REDUCE(nid $m, [d_1, d_2, \ldots]$)
        $d_{min} \leftarrow \infty$
        $M \leftarrow \emptyset$
        for all $d \in$ counts $[d_1, d_2, \ldots]$ **do**
            if ISNODE(d) **then**
                $M \leftarrow d$ ▷ Recover graph structure
            else if $d < d_{min}$ **then** ▷ Look for shorter distance
                $d_{min} \leftarrow d$
        M.DISTANCE $\leftarrow d_{min}$ ▷ Update shortest distance
        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 any step it only pursues edges from the minimum-cost path inside the 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?

Weighted Edges

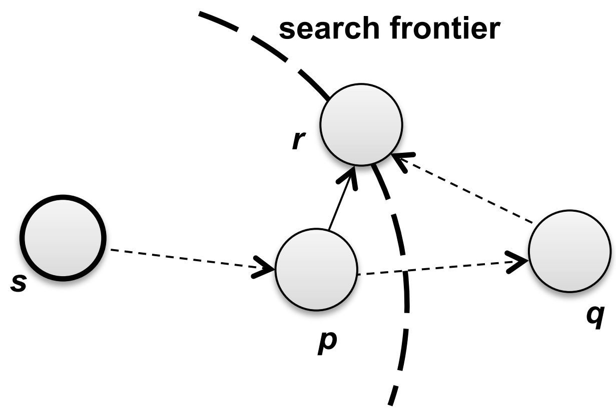
- Now add positive weights to the edges
  - Why can't edge weights be negative?
- Simple change: adjacency list now includes a weight w for each edge
  - In mapper, emit (m, $d + w_p$) instead of (m, $d + 1$) for each node m
- That's it?

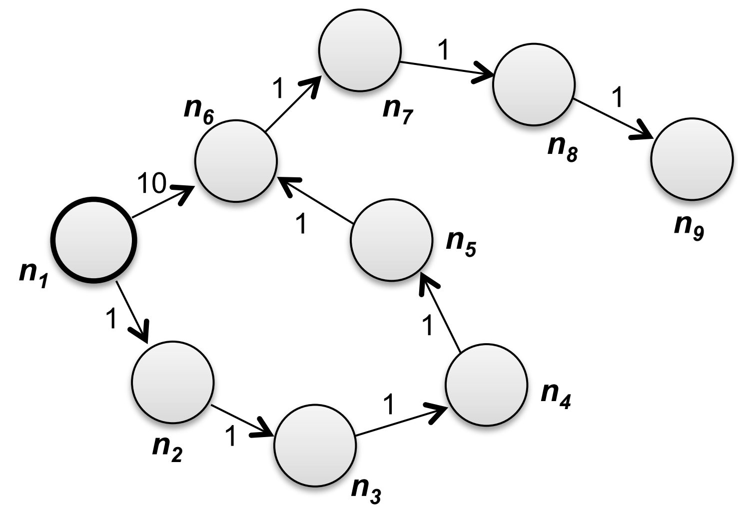
Stopping Criterion

- 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

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
- 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

Connection to Theory

- Bulk Synchronous Processing (1990 Valiant)
- Nodes (Processors) can communicate with any neighbor
- However, messages do not arrive until synchronization phase

Random Walks Over the Web

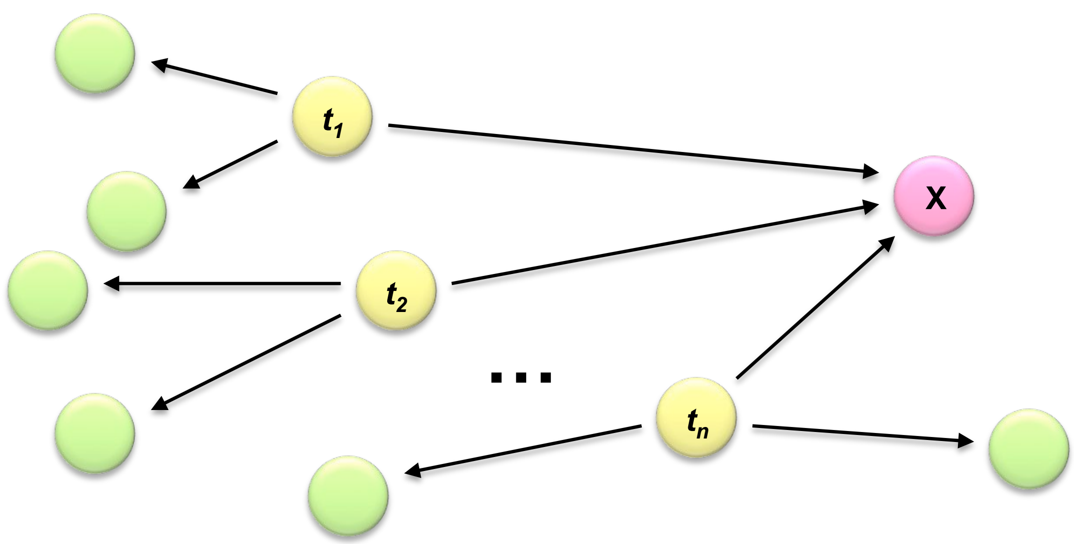
- Random surfer model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- PageRank
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages
- PageRank captures notions of page importance
  -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

PageRank: Defined

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

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



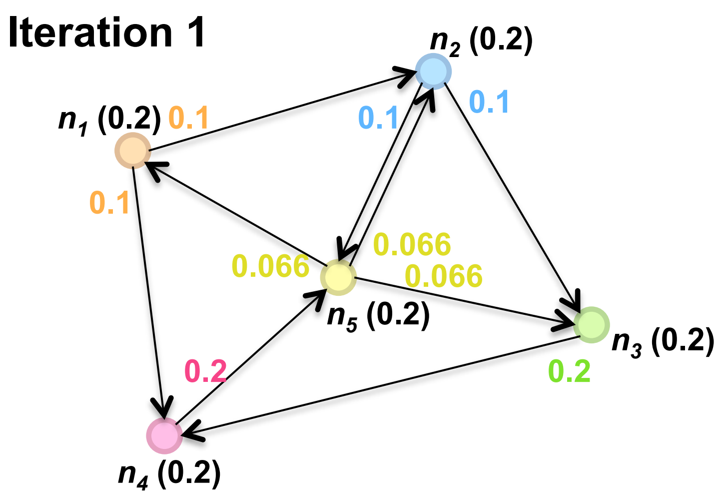
Computing PageRank

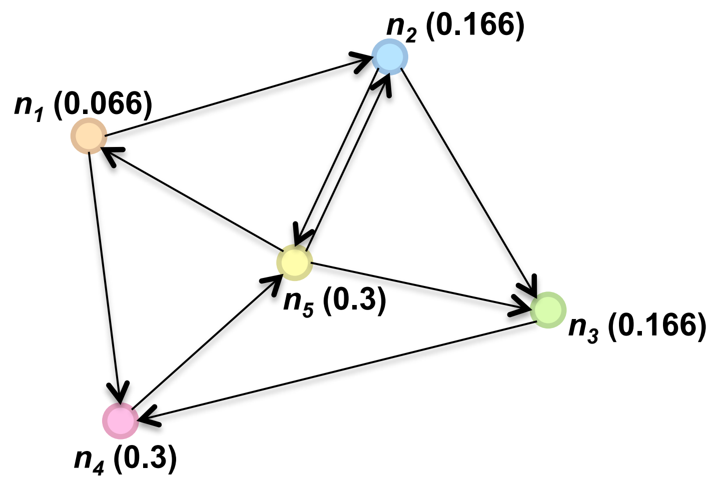
- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm:
  - Start with seed PR_i values
  - Each page distributes PR_i “credit” to all pages it links to
  - Each target page adds up “credit” from multiple in-bound links to compute PR_{i+1}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

Simplified PageRank

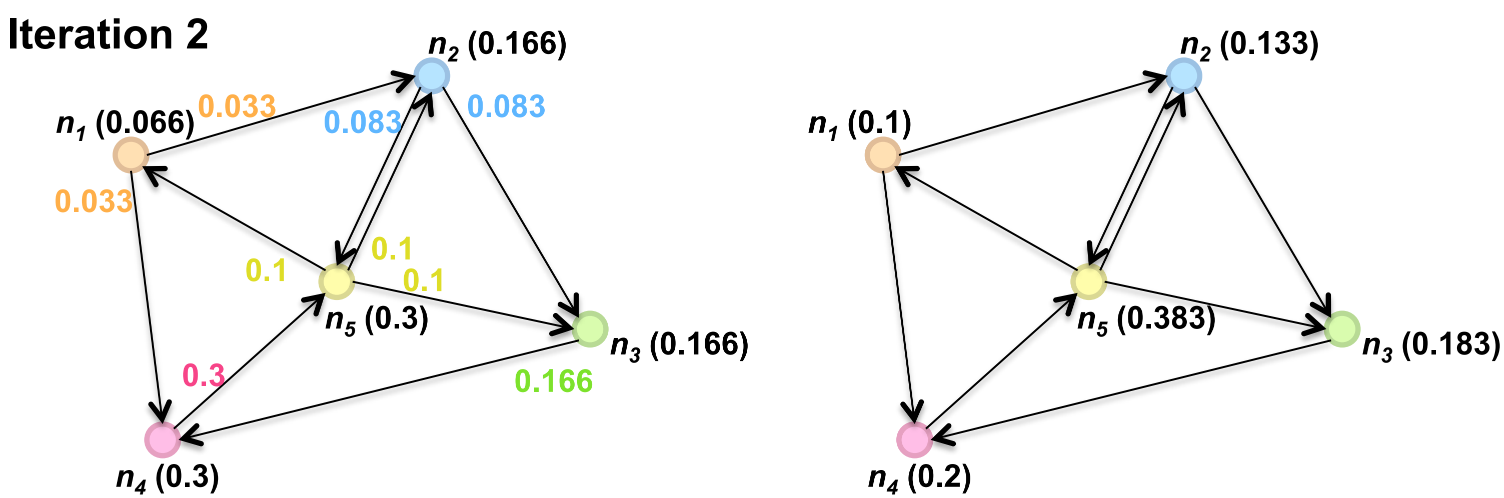
- First, tackle the simple case:
  - No random jump factor
  - No dangling links
- Then, factor in these complexities…
  - Why do we need the random jump?
  - Where do dangling links come from?

Sample PageRank Iteration (1)

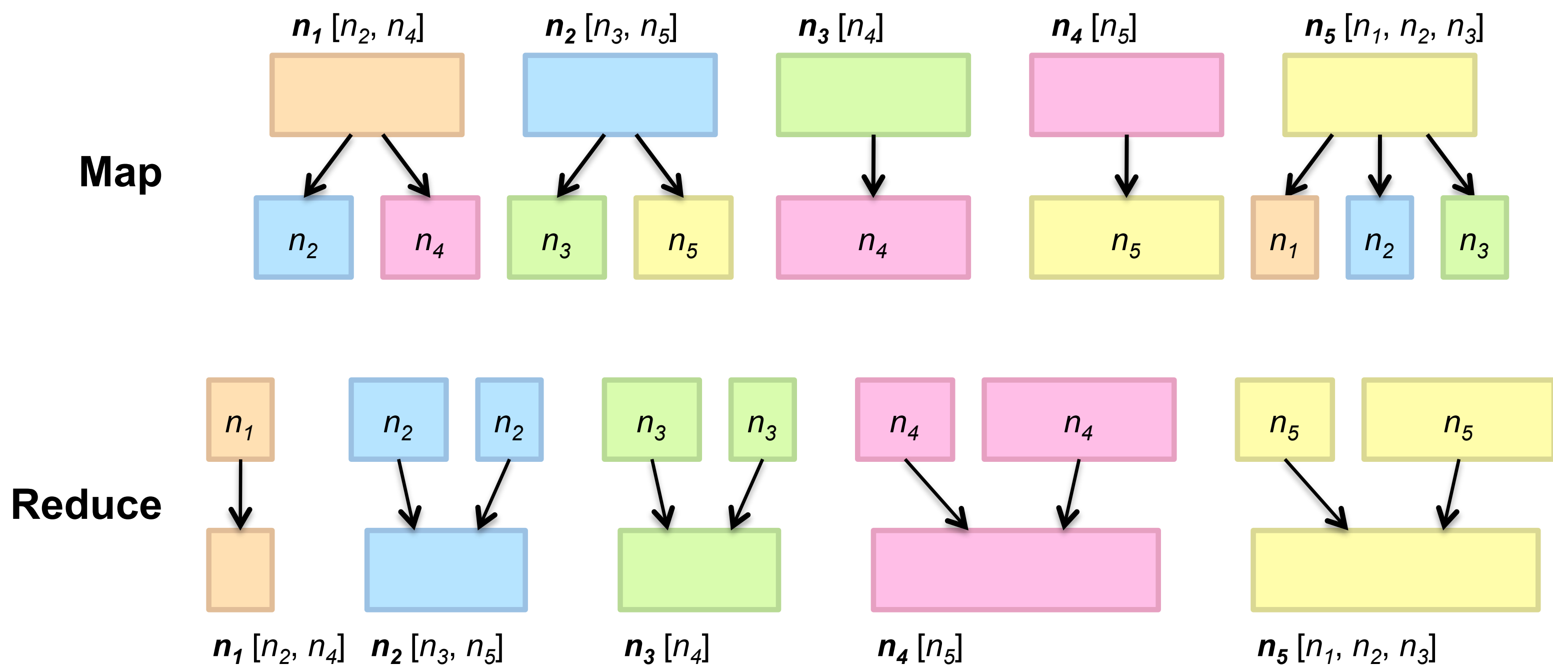




Sample PageRank Iteration (2)



PageRank in MapReduce



PageRank Pseudo-Code

```
class Mapper
    method Map(nid n, node N)
        p ← N.PageRank/|N.AdjacencyList|
        Emit(nid n, N)                              ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, p)                          ▷ Pass PageRank mass to neighbors

class Reducer
    method Reduce(nid m, [p1, p2, ...])
        M ← ∅
        for all p ∈ counts [p1, p2, ...] do
            if IsNode(p) then
                M ← p                               ▷ Recover graph structure
            else
                s ← s + p                           ▷ Sums incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```

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
- Convergence for web graphs?

Beyond PageRank

- Link structure is important for web search
  - PageRank is one of many link-based features: HITS, SALSA, etc.
  - One of many thousands of features used in ranking…
- Adversarial nature of web search
  - Link spamming
  - Spider traps
  - Keyword (Language Model) stuffing
  - Domain Sniping
  - Requester-Mirage
  - …

Digging In: Counters

- How do you know how many dangling pages?
- Use counters
  - Many built in counters
  - Visible on JobTracker
  - Keeps long-running jobs from being killed
  - Good for debugging

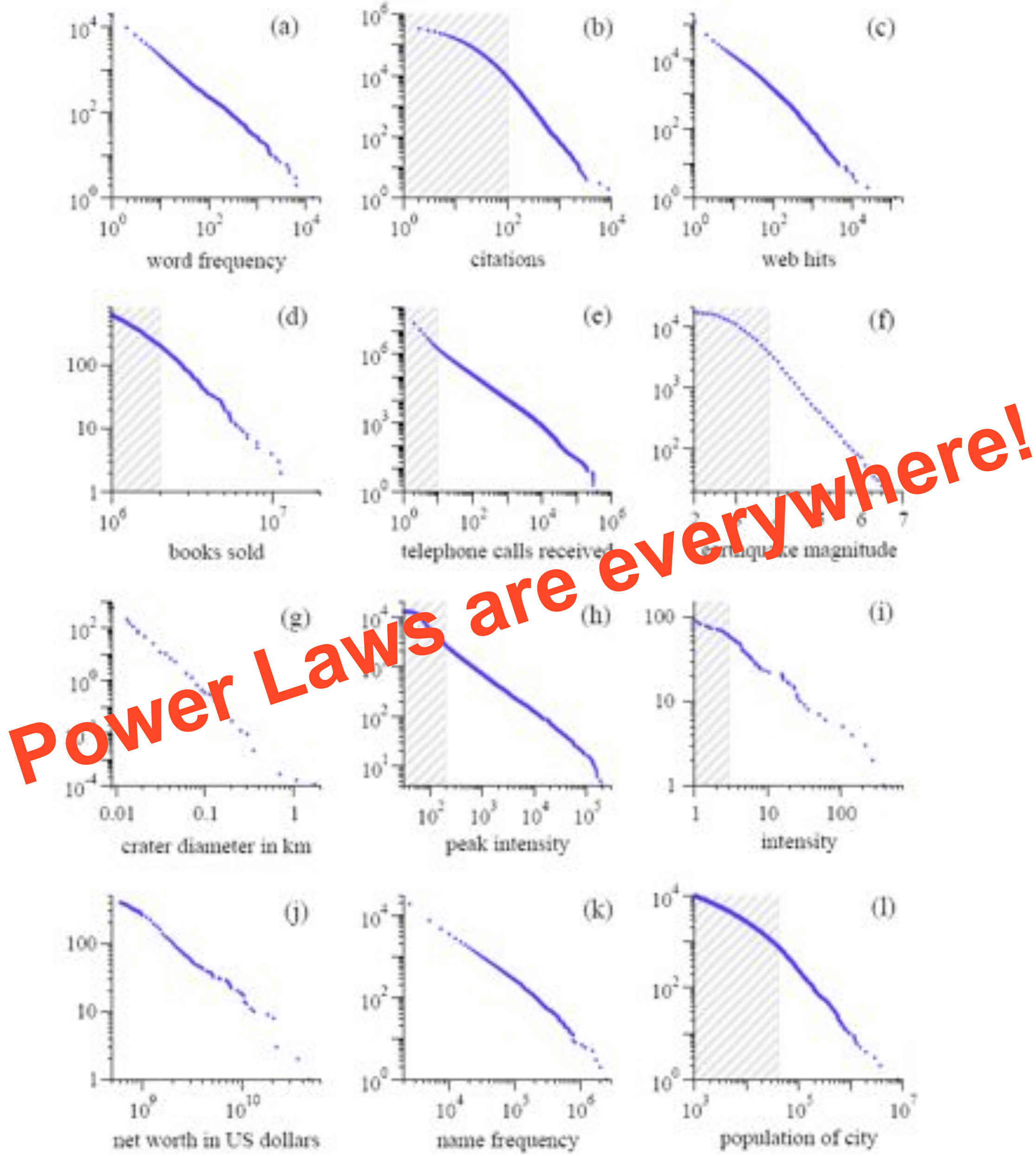
```
static enum WordType {
  STARTS_WITH_DIGIT,
  STARTS_WITH_LETTER
}
```

```
context.getCounter(WordType.STARTS_WITH_LETTER).increment(1);
```

```
RunningJob job = JobClient.runJob(conf);  // blocks until job completes
Counters c = job.getCounters();
long cnt = c.getCounter(WordType.STARTS_WITH_DIGIT);
```

Efficient Graph Algorithms

- Sparse vs. dense graphs
- Graph topologies



Power Laws are everywhere!

Figure from: Newman, M. E. J. (2005) "Power laws, Pareto distributions and Zipf's law." Contemporary Physics 46:323–351.

Local Aggregation

- Use combiners!
  - In-mapper combining design pattern also applicable
- Maximize opportunities for local aggregation
  - Simple tricks: sorting the dataset in specific ways
  - Partition graphs

Graphs at Google

- MapReduce – designed to handle PageRank
- MapReduce still handles 80% of computations
- Pregel (based on BSP)
  - Node – centric computation
    - Can send messages to neighbors
    - Can add edges, neighbors
    - Process previous messages
  - Handle conflict
  - Provide partitioning heuristics (reduce communication)
  - Not public

Source: Wikipedia (Japanese rock garden)

Digging In: BFS Node

```
public class BFSNode implements Writable {

  public static final int TYPE_COMPLETE = 1;
  public static final int TYPE_DISTANCE = 2;
  public static final int TYPE_STRUCTURE = 3;

  private int mType;
  private int mNodeId;
  private int mDistance;
  private ArrayListOfIntsWritable mAdjacenyList;

  public BFSNode() {
  }

  public int getDistance() {
    return mDistance;
  }

  public void setDistance(int d) {
    mDistance = d;
  }

  public int getNodeId() {
    return mNodeId;
  }

  public void setNodeId(int n) {
    mNodeId = n;
  }

  public ArrayListOfIntsWritable getAdjacenyList() {
    return mAdjacenyList;
  }
```

Digging In: BFS Node

```
public void readFields(DataInput in) throws IOException {
  mType = in.readByte();

  mNodeId = in.readInt();

  if (mType == TYPE_DISTANCE) {
    mDistance = in.readInt();
    return;
  }

  if (mType == TYPE_COMPLETE) {
    mDistance = in.readInt();
  }

  mAdjacenyList = new ArrayListOfIntsWritable();
  mAdjacenyList.readFields(in);
}
```

Digging In: BFS Node

```
public void write(DataOutput out) throws IOException {
  out.writeByte((byte) mType);
  out.writeInt(mNodeId);

  if (mType == TYPE_DISTANCE) {
    out.writeInt(mDistance);
    return;
  }

  if (mType == TYPE_COMPLETE) {
    out.writeInt(mDistance);
  }

  mAdjacenyList.write(out);
}
```

Digging In: BFS Node

```java
public String toString() {
  StringBuilder s = new StringBuilder();

  s.append("{");
  s.append(mNodeId);

  s.append(" ");
  s.append(mDistance);
  s.append(" ");

  if (mAdjacenyList == null) {
    s.append("[]");
  } else {
    s.append("[");
    for (int i = 0; i < mAdjacenyList.size(); i++) {
      s.append(mAdjacenyList.get(i));
      if (i < mAdjacenyList.size() - 1)
        s.append(", ");
    }
    s.append("]");
  }

  s.append(" }");

  return s.toString();
}
```

Digging In: BFS Mapper

```
// Mapper with in-mapper combiner optimization.
private static class MapClass extends
    Mapper<IntWritable, BFSNode, IntWritable, BFSNode> {

  // For buffering distances keyed by destination node.
  private static final HMapII map = new HMapII();

  // For passing along node structure.
  private static final BFSNode intermediateStructure = new BFSNode();
```

Digging In: BFS Mapper

```java
@Override
public void map(IntWritable nid, BFSNode node, Context context) throws IOException,
    InterruptedException {

  // Pass along node structure.
  intermediateStructure.setNodeId(node.getNodeId());
  intermediateStructure.setType(BFSNode.TYPE_STRUCTURE);
  intermediateStructure.setAdjacencyList(node.getAdjacenyList());

  context.write(nid, intermediateStructure);

  if (node.getDistance() == Integer.MAX_VALUE) {
    return;
  }
```

Digging In: BFS Mapper

```
    context.getCounter(ReachableNodes.Map).increment(1);
    // Retain distance to self.
    map.put(nid.get(), node.getDistance());

    ArrayListOfInts adj = node.getAdjacenyList();
    int dist = node.getDistance() + 1;
    // Keep track of shortest distance to neighbors.
    for (int i = 0; i < adj.size(); i++) {
      int neighbor = adj.get(i);

      // Keep track of distance if it's shorter than previously
      // encountered, or if we haven't encountered this node.
      if ((map.containsKey(neighbor) && dist < map.get(neighbor)) ||
          !map.containsKey(neighbor)) {
        map.put(neighbor, dist);
      }
    }
  }
```

Digging In: BFS Mapper

```java
  @Override
  public void cleanup(Mapper<IntWritable, BFSNode, IntWritable, BFSNode>.Context context)
      throws IOException, InterruptedException {
    // Now emit the messages all at once.
    IntWritable k = new IntWritable();
    BFSNode dist = new BFSNode();

    for (MapII.Entry e : map.entrySet()) {
      k.set(e.getKey());

      dist.setNodeId(e.getKey());
      dist.setType(BFSNode.TYPE_DISTANCE);
      dist.setDistance(e.getValue());

      context.write(k, dist);
    }
  }
}
```

Digging In: BFS Reducer

```java
@Override
public void reduce(IntWritable nid, Iterable<BFSNode> iterable, Context context)
    throws IOException, InterruptedException {

  Iterator<BFSNode> values = iterable.iterator();

  int structureReceived = 0;
  int dist = Integer.MAX_VALUE;
  while (values.hasNext()) {
    BFSNode n = values.next();

    if (n.getType() == BFSNode.TYPE_STRUCTURE) {
      // This is the structure; update accordingly.
      ArrayListOfIntsWritable list = n.getAdjacenyList();
      structureReceived++;

      int arr[] = new int[list.size()];
      for (int i = 0; i < list.size(); i++) {
        arr[i] = list.get(i);
      }

      node.setAdjacencyList(new ArrayListOfIntsWritable(arr));
    } else {
      // This is a message that contains distance.
      if (n.getDistance() < dist) {
        dist = n.getDistance();
      }
    }
  }
}
```

Digging In: BFS Reducer

```
    node.setType(BFSNode.TYPE_COMPLETE);
    node.setNodeId(nid.get());
    node.setDistance(dist); // Update the final distance.

    if (dist != Integer.MAX_VALUE) {
      context.getCounter(ReachableNodes.Reduce).increment(1);
    }

    // Error checking.
    if (structureReceived == 1) {
      // Everything checks out, emit final node structure with updated
      // distance.
      context.write(nid, node);
    } else if (structureReceived == 0) {
      // We get into this situation if there exists an edge pointing
      // to a node which has no corresponding node structure (i.e.,
      // distance was passed to a non-existent node)... log but move
      // on.
      LOG.warn("No structure received for nodeid: " + nid.get());
    } else {
      // This shouldn't happen!
      throw new RuntimeException("Multiple structure received for nodeid: " + nid.get()
          + " struct: " + structureReceived);
    }
  }
}
```

Digging In: Runner

- For multiple iterations, use multiple jobs inside a for loop
- Convergence?
- Combiner?

Source: Wikipedia (Japanese rock garden)