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
  - ...
Some Graph Problems

- Finding shortest paths
  - Routing Internet traffic and UPS trucks
- Finding minimum spanning trees
  - Telecommunication company 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

Representing Graphs

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

Represent a graph as an \( n \times n \) square matrix \( M \)
- \( n = |V| \)
- \( M_{ij} = 1 \) means a link from node \( i \) to \( j \)

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & 1 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 \\
3 & 1 & 0 & 0 & 0 \\
4 & 1 & 0 & 1 & 0 \\
\end{array}
\]

Adjacency Lists

Take adjacency matrices... and throw away all the zeros

1: 2, 4
2: 1, 3, 4
3: 1
4: 1, 3
Single Source Shortest Path

- **Problem:** find shortest path from a source node to target nodes
- **Dijkstra’s Algorithm**
  - Using a priority queue, it is the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights
  - Priority queue restricts utilization to single threaded machine
- **Breadth-First Search (BFS)**
  - considers outgoing edges of the vertex's predecessor in the search, before any outgoing edges of the vertex

Breadth-First Search (BFS)

- Assuming equal edge weights: first time you see a vertex you found the minimal distance!
Breadth-first search

```java
foreach ( vertex u ∈ V[G] ) {
    color[u] = white;
    dist[u] = ∞;
    prev[u] = NIL;
}
color[s] = gray;
dist[s] = 0;
Q.add(s);
while ( Q.notempty() ) {
    u = Q.dequeue();
    foreach ( v ∈ adj[u] ){
        if ( color[v] == white ) {
            color[v] = gray;
            dist[v] = d[u] + 1;
            prev[v] = u;
            Q.add(v);
        }
    }
    color[u] = black
}
```

Algorithm shown here determines the minimum distance of all vertices connected to a source vertex s, assuming equal edge weights.

**Breadth-first search**

- Extending sequential breadth-first search to MapReduce:
  - `DISTANCETo(startNode) = 0`
  - For all nodes n directly reachable from startNode, `DISTANCETo(n) = 1`
  - For all nodes n reachable from some other set of nodes S, `DISTANCETo(n) = 1 + min(DISTANCETo(m), m ∈ S)`
From Intuition to Algorithm

- **Mapper input**
  - Key: node $n$
  - Value: $D$ (distance from start), adjacency list (list of nodes reachable from $n$)

- **Mapper output**
  - $\forall p \in$ targets in adjacency list:
    - emit (key = $p$, value = $D+1$)

- The reducer gathers possible distances to a given $p$ and selects the minimum one
  - Additional bookkeeping needed to keep track of actual path, e.g.
    - emit( key = $p$, value = \{D+1, $n$\} )

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
  - Feed output back into the same MapReduce task

- Preserving graph structure:
  - Problem: Where did the adjacency list go?
  - Solution: mapper emits ($n$, adjacency list) as well

- Simple change: adjacency list in map task includes a weight $w$ for each edge
  - emit ($p$, $D+w_p$) instead of ($p$, $D+1$) for each node $p$
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
8.     else if d < d_min then
9.         d_min ← d
10.    M.DISTANCE ← d_min
11.   Emit(nid m, node M)

GenericWritables

• In the previous algorithm, Mapper has to emit two different types of values: Adjacency list and Distance
• Solution: extending a GenericWritable class

  public class MultiValueWritable extends GenericWritable{
      private static Class[] CLASSES = new Class[] {
          IntWritable.class,
          Text.class
      }
      ...

• And in reducer need to check for the type of the value

  public void reduce (Text key, Iterable<MultiValueWritable> vals){
      for (MultiValueWritable mv : values ){
          if ( mv instanceof IntWritable ) {
              ...
          }
      }
  }
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 in-bound links \( t_1, \ldots, t_n \), where

- \( C(t) \) is the out-degree of \( t \)
- \( \alpha \) is probability of random jump
- \( N \) is the total number of nodes in the graph

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

- Focusing on the second part of the formula (i.e. \( \alpha=0 \)
Sample PageRank Iteration (1)

- Starting point: all vertices get the same rank (e.g., 0.2)
- Determine for each outgoing edge the contribution of that weight, i.e., \( \text{rank} / \text{no. of outgoing edges} \)
- Determine the new rank of a vertex by adding up the partial incoming ranks from all connected vertices

Sample PageRank Iteration (2)
PageRank Pseudo-Code

1: class Mapper
2: method Map(nid, node N)
3: \( p \leftarrow N.\text{PageRank}[N.\text{AdjacencyList}] \)
4: Emit(nid, N)
5: for all nodeid \( m \in N.\text{AdjacencyList} \) do
6: \( \text{Emit}(\text{nid}, m, p) \) \( \triangleright \) Pass PageRank mass to neighbors

1: class Reducer
2: method Reduce(nid \( m, [p_1, p_2, \ldots] \))
3: \( M \leftarrow \emptyset \)
4: for all \( p \in \text{counts} [p_1, p_2, \ldots] \) do
5: \( \text{if IsNode}(p) \) then
6: \( M \leftarrow p \)
7: else
8: \( s \leftarrow s + p \) \( \triangleright \) Sums incoming PageRank contributions
9: \( M.\text{PageRank} \leftarrow s \)
10: Emit(nid \( m, \text{node} M \))

PageRank Convergence

- Convergence criteria
  - Iterate until PageRank values don’t change
  - Iterate until PageRank rankings don’t change
  - Fixed number of iterations
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

Google Pregel

- Literature: G. Malewicz, M.H. Austern, A.J.C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A System for Large-Scale Graph Processing” [link]
- Distributed system developed for large scale graph processing
- Intuitive API: ‘think like a vertex, not a key-value pair’
- Bulk Synchronous Parallel (BSP) as execution model
- fault tolerance by checkpointing
- Pregel is proprietary, but:
  - Apache Giraph is an open source implementation of Pregel
  - runs on standard Hadoop infrastructure
  - computation is executed in memory
Bulk Synchronous Parallel (BSP) Execution Model

- parallel computing model that can be used to guide the design, analysis and implementation of parallel algorithms
- takes communication and computation into account

Vertex-centric BSP

- Every vertex has an id, a value, a list of its adjacent vertex ids and the corresponding edge values
- Every vertex is invoked in every superstep, can recompute its value and send messages to other vertices, which are delivered over superstep barriers
- Advanced features: termination votes, combiners, aggregators, topology mutations
Apache Giraph

- **Master**: responsible for **coordination**
  - assigns partitions to workers
  - coordinates synchronization
  - requests checkpoints
  - aggregates aggregator values
  - collects health statuses

- **Worker**: responsible for **vertices**
  - invokes active vertices compute() function
  - sends, receives and assigns messages
  - computes local aggregation values

- **ZooKeeper**: responsible for **computation state**
  - partition/worker mapping
  - global state: #superstep
  - checkpoint paths, aggregator values, statistics

---

**What you have to implement**

- Your algorithm as a Vertex
  - Subclass one of the many existing implementations: `BasicVertex`, `MutableVertex`, `EdgeListVertex`, `HashMapVertex`, `LongDoubleFloatDoubleVertex`, ...

- A VertexInputFormat to read your graph
  - e.g. from a text file with adjacency lists like `<vertex> <neighbor1> <neighbor2> ...

- A VertexOutputFormat to write back the result
  - e.g. `<vertex> <pageRank>`
public void compute(Iterable<IntWritable> messages)
    throws IOException {
    if (getSuperstep() == 0) {
        setValue(MAX_INT);
        if (isSource()) {
            for (Edge<Text, NullWritable> e : getEdges()) {
                sendMessage(e.getTargetVertexId(), msg);
            }
        }
    }
    int min = getValue().get();
    for (IntWritable msg : messages) {
        min = Math.min(msg.get(), min);
    }
    if (min < getValue().get()) {
        setValue(new IntWritable(min));
        msg.set(min + 1);
        sendMessageToAllEdges(msg);
    }
    voteToHalt();
}

• Why not implement Giraph with multiple MapReduce jobs
  - each iteration becomes a MapReduce job
  - too much disk traffic, no in-memory caching
• Giraph is a single Map-only job in Hadoop
  - Hadoop is purely a resource manager for Giraph
  - Communication is done through Netty-based IPC
• Combiners: users can implement a combine() method that can reduce the amount of messages sent and received
• Aggregators: commutative and associative operations that are performed globally (similarly to MPI)