Let us now look at implementing graph algorithms in MapReduce.

Why Graphs?

• Discussion is based on the book and slides by Jimmy Lin and Chris Dyer
• Analyze hyperlink structure of the Web
• Social networks
  – Facebook friendships, Twitter followers, email flows, phone call patterns
• Transportation networks
  – Roads, bus routes, flights
• Interactions between genes, proteins, etc.
What is a Graph?

• $G = (V, E)$
  – $V$: set of vertices (nodes)
  – $E$: set of edges (links), $E \subseteq V \times V$
• Edges can be directed or undirected
• Graph might have cycles or not (acyclic graph)
• Nodes and edges can be annotated
  – E.g., social network: node has demographic information like age; edge has type of relationship like friend or family

Graph Problems

• Graph search and path planning
  – Find driving directions from A to B
  – Recommend possible friends in social network
  – How to route IP packets or delivery trucks
• Graph clustering
  – Identify communities in social networks
  – Partition large graph to parallelize graph processing
• Minimum spanning trees
  – Connected graph of minimum total edge weight
More Graph Problems

• Bipartite graph matching
  – Match nodes on “left” with nodes on “right” side
  – E.g., match job seekers and employers, singles looking for dates, papers with reviewers

• Maximum flow
  – Maximum traffic between source and sink
  – E.g., optimize transportation networks

• Finding “special” nodes
  – E.g., disease hubs, leader of a community, people with influence

Graph Representations

• Usually one of these two:
  – Adjacency matrix
  – Adjacency list
Adjacency Matrix

• Matrix $M$ of size $|N| \times |N|$
  – Entry $M(i,j)$ contains weight of edge from node $i$ to node $j$; 0 if no edge

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Example source: Jimmy Lin

Properties

• Advantages
  – Easy to manipulate with linear algebra
  – Operation on outlinks and inlinks corresponds to iteration over rows and columns

• Disadvantage
  – Huge space overhead for sparse matrix
  – E.g., Facebook friendship graph
Adjacency List

- Compact row-wise representation of matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

Properties

- Advantages
  - More space-efficient
  - Still easy to compute over outlinks for each node
- Disadvantage
  - Difficult to compute over inlinks for each node

- Note: remember inverse Web graph discussion
Parallel Breadth-First Search

- Case study: single-source shortest path problem
  - Find the shortest path from a source node $s$ to all other nodes in the graph
- For non-negative edge weights, Dijkstra’s algorithm is the classic sequential solution
  - Initialize distance $d[s]=0$, all others to $\infty$
  - Maintain priority queue of nodes sorted by distance
  - Remove first node $u$ from queue and update $d[v]$ for each node $v$ in adjacency list of $u$ if (1) $v$ is in queue and (2) $d[v] > d[u] + \text{weight}(u,v)$

Dijkstra’s Algorithm Example

Example from Jimmy Lin’s presentation
Dijkstra’s Algorithm Example

Dijkstra’s Algorithm Example
Dijkstra’s Algorithm Example

![Graph Diagram]

Example from CLR

Dijkstra’s Algorithm Example

![Graph Diagram]
Parallel Single-Source Shortest Path

- Priority queue is core element of Dijkstra’s algorithm
  - No global shared data structure in MapReduce
- Dijkstra’s algorithm proceeds sequentially, node by node
  - Taking non-min node could affect correctness of algorithm
- Solution: perform parallel breadth-first search
Parallel Breadth-First Search

• Start at source s
• In first round, find all nodes reachable in one hop from s
• In second round, find all nodes reachable in two hops from s, and so on
• Keep track of min distance for each node
  – Also record corresponding path
• Iterations stop when no shorter path possible

BFS Visualization

Example from Jimmy Lin's presentation
MapReduce Code: Single Iteration

```java
map(nid n, node N) // N stores node’s current min distance and adjacency list
d = N.distance
emit(nid n, N) // Pass along graph structure
for all nid m in N.adjacencyList do
    emit(nid m, d + w(n,m)) // Emit distances to reachable nodes

reduce(nid m, [d1,d2,...])
dMin = ∞; M = ∅
for all d in [d1,d2,...] do
    if isNode(d) then
        M = d // Recover graph structure
    else if d < dMin then
        dMin = d // Look for min distance in list
        M.distance = dMin // Update node’s shortest distance
    emit(nid m, node M)
```

Overall Algorithm

- Need driver program to control the iterations
- Initialization: `SourceNode.distance = 0`, all others have distance=\(\infty\)
- When to stop iterating?
  - If all edges have weight 1, can stop as soon as no node has \(\infty\) distance any more
    - Can detect this with Hadoop counter
  - Number of iterations depends on graph diameter
    - In practice, many networks show the small-world phenomenon, e.g., six degrees of separation
Dealing With Diverse Edge Weights

- “Detour” path can be shorter than “direct” connection, hence cannot stop as soon as all node distances are finite
- Stop when no node’s shortest distance changes any more
  - Can be detected with Hadoop counter
  - Worst case: $|N|$ iterations

MapReduce Algorithm Analysis

- Brute-force approach that performs many irrelevant computations
  - Computes distances for nodes that still have infinity distance
  - Repeats previous computations inside “search frontier”
- Dijkstra’s algorithm only explores the search frontier, but needs the priority queue
Typical Graph Processing in MapReduce

- Graph represented by adjacency list per node, plus extra node data
- Map works on a single node u
  - Node u’s local state and links only
- Node v in u’s adjacency list is intermediate key
  - Passes results of computation along outgoing edges
- Reduce combines partial results for each destination node
- Map also passes graph itself to reducers
- Driver program controls execution of iterations

PageRank Introduction

- Popularized by Google for evaluating the quality of a Web page
- Based on random Web surfer model
  - Web surfer can reach a page by jumping to it or by following the link from another page pointing to it
  - Modeled as random process
- Intuition: important pages are linked from many other (important) pages
  - Goal: find pages with greatest probability of access
PageRank Definition

- PageRank of page n:
  \[ P(n) = \alpha \frac{1}{|V|} + (1 - \alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)} \]
  - \(|V|\) is number of pages (nodes)
  - \(\alpha\) is probability of random jump
  - \(L(n)\) is the set of pages linking to n
  - \(P(m)\) is m’s PageRank
  - \(C(m)\) is m’s out-degree
- Definition is recursive
  - Compute by iterating until convergence (fixpoint)

Computing PageRank

- Similar to BFS for shortest path
- Computing \(P(n)\) only requires \(P(m)\) and \(C(m)\) for all pages linking to n
  - During iteration, distribute \(P(m)\) evenly over outlinks
  - Then add contributions over all of n’s inlinks
- Initialization: any probability distribution over the nodes
PageRank Example

Iteration 1

Source: Jimmy Lin's presentation

PageRank Example

Iteration 2
PageRank in MapReduce

MapReduce Code

map(nid n, node N)  // N stores node's current PageRank and adjacency list
p = N.pageRank / |N.adjacencyList|
emit(nid n, N)  // Pass along graph structure
for all nid m in N.adjacencyList do
    emit(nid m, p)  // Pass PageRank mass to neighbors

reduce(nid m, [p1,p2,...])
s=0; M = ∅
for all p in [p1,p2,...] do
    if isNode(p) then
        M = p  // Recover graph structure
    else
        s += p  // Sum incoming PageRank contributions
M.pageRank = α/|V| + (1-α)·s
emit(nid m, node M)
Dangling Nodes

• Consider node $x$ with no outgoing links
  – $P(x)$ is not passed to any other node, hence gets “lost” in the Map phase
• Need to correct for the missing probability mass
  – Model: assume dangling page links to all pages
  – Mathematically equivalent to
    $$P(n) = \alpha \frac{1}{|V|} + (1 - \alpha) \left( \frac{\delta}{|V|} + \sum_{m \in L(n)} \frac{P(m)}{C(m)} \right)$$
  – $\delta$: missing PageRank mass due to dangling nodes

PageRank with Dangling Nodes

• Challenge: need $\delta$, which is the sum over the current page ranks of dangling nodes
  – MR-job1: compute $\delta$
  – MR-job2: compute new PageRank using $\delta$
• Alternative computations?
  – Order inversion pattern to make sure $\delta$ is available in all reduce tasks
Number of Iterations

- PageRank computation iterates until convergence
  - PageRank of all nodes no longer changes (or is within small tolerance)
  - Needs to be checked by driver
- Original PageRank paper: 52 iterations until convergence on graph with 322 million edges
  - Highly dependent on data properties

General Graph Processing Issues

- Sequential algorithms often use **global** data structure for efficiency
- In MapReduce with adjacency list representation, information can only be passed **locally** to or from direct neighbors
  - But can pre-compute other data structures, e.g., two-hop neighbors
- Presented algorithms have Map output of $O(#\text{edges})$, which works well for sparse graphs
General Graph Processing Issues

• Partitioning of graph into chunks strongly affects effectiveness of combiners
  – Often best to keep well-connected components together
• Numerical stability for large graphs
  – PageRank of individual page might be so small that it underflows standard floating point representation
  – Can work with logarithm-transformed numbers instead