Data-Intensive Distributed Computing
CS 431/631 451/651 (Winter 2021)

Part 5: Analyzing Graphs (1/2)

Ali Abedi

These slides are available at https://www.student.cs.uwaterloo.ca/~cs451/

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details
Structure of the Course

- Analyzing Text
- Analyzing Graphs
- Analyzing Relational Data
- Data Mining

“Core” framework features and algorithm design
What’s a graph?

\[ G = (V,E), \text{ where} \]

- \( V \) represents the set of vertices (nodes)
- \( E \) represents the set of edges (links)
- Edges may be directed or undirected
- Both vertices and edges may contain additional information
Examples of Graphs

- Social networks
- Hyperlink structure of the web
- Computers on the Internet

We’re mostly interested in sparse graphs!
Representing Graphs

- Adjacency matrices
- Adjacency lists
- Edge lists
Adjacency Matrices

Represent a graph as an $n \times n$ square matrix $M$

$n = |V|$

$M_{ij} = 1$ iff an edge from vertex $i$ to $j$

<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>
Adjacency Matrices: Critique

Advantages
Amenable to mathematical manipulation
Intuitive iteration over rows and columns

Disadvantages
Lots of wasted space (for sparse matrices)
Adjacency Lists

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

We have seen this in posting lists.
Adjacency Lists: Critique

Advantages
Much more compact representation (compress!)
Easy to compute over outlinks

Disadvantages
Difficult to compute over inlinks
Edge Lists

Explicitly enumerate all edges

<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)  
(1, 4)  
(2, 1)  
(2, 3)  
(2, 4)  
(3, 1)  
(4, 1)  
(4, 3)
Edge Lists: Critique

Advantages
Easily support edge insertions

Disadvantages
Wastes spaces
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
 Halting the spread of avian flu

Bipartite matching
 match.com

Web ranking
 PageRank
What does the web look like?

Analysis of a large webgraph from the common crawl: 3.5 billion pages, 129 billion links
Meusel et al. Graph Structure in the Web — Revisited. WWW 2014.
What does the web look like?
Very roughly, a scale-free network

Fraction of $k$ nodes having $k$ connections:

$$P(k) \sim k^{-\gamma}$$

(i.e., degree distribution follows a power law)
How do we extract the webgraph?
The webgraph... is big?!

webgraph from the common crawl: 3.5 billion pages, 129 billion links
Meusel et al. Graph Structure in the Web — Revisited. WWW 2014.

58 GB!
Graphs and MapReduce (and Spark)

A large class of graph algorithms involve:
Local computations at each node
Propagating results: “traversing” the graph

Key questions:
How do you represent graph data in MapReduce (and Spark)?
How do you traverse a graph in MapReduce (and Spark)?
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
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:

Define: \( b \) is reachable from \( a \) if \( b \) is on adjacency list of \( a \)

\[
\text{DISTANCETo}(s) = 0
\]

For all nodes \( p \) reachable from \( s \),

\[
\text{DISTANCETo}(p) = 1
\]

For all nodes \( n \) reachable from some other set of nodes \( M \),

\[
\text{DISTANCETo}(n) = 1 + \min(\text{DISTANCETo}(m), m \in M)
\]
Visualizing Parallel BFS
From Intuition to Algorithm

Data representation:
- Key: node $n$
- Value: $d$ (distance from start), adjacency list
- 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 “frontier” by one hop
Subsequent iterations include 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 \((n, \text{adjacency list})\) as well

Ugh! This is ugly!
BFS Pseudo-Code

class Mapper {
    def map(id: Long, n: Node) = {
        emit(id, n) // emit graph structure
        val d = n.distance
        for (m <- n.adjacencyList) {
            emit(m, d+1)
        }
    }
}

class Reducer {
    def reduce(id: Long, objects: Iterable[Object]) = {
        var min = infinity
        var m = null
        for (d <- objects) {
            if (isNode(d)) m <- d
            else if d < min min = d
        }
        m.distance = min
        emit(id, m)
    }
}
Stopping Criterion  
(equal edge weight)

How many iterations are needed in parallel BFS?

Convince yourself: when a node is first “discovered”, we’ve found the shortest path

What does it have to do with six degrees of separation?
Frontier size during BFS traversal
Implementation Practicalities

Convergence?

HDFS

map
reduce

HDFS
Comparison to Dijkstra

Dijkstra’s algorithm is more efficient
At each step, only pursues edges from minimum-cost path inside 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?

We can’t do better because we cannot keep a global state like Dijkstra does.
Single Source: Weighted Edges

Now add positive weights to the edges
Simple change: add weight $w$ for each edge in adjacency list

Simple change: add weight $w$ for each edge in adjacency list
In mapper, emit $(m, d + w_p)$ instead of $(m, d + 1)$ for each node $m$

That’s it?
Stopping Criterion
(positive edge weight)

How many iterations are needed in parallel BFS?

Convince yourself: when a node is first “discovered”,
we’ve found the shortest path

Not true!
Additional Complexities

![Diagram](image)

- s
- search frontier
- r
- p
- q
- n₁
- n₂
- n₃
- n₄
- n₅
- n₆
- n₇
- n₈
- n₉
- n₁₀