Distributed data processing on the Cloud – Lecture 5

Graph Data Processing with MapReduce

Satish Srirama

Some material adapted from slides by Jimmy Lin, 2015 (licensed under Creation Commons Attribution 3.0 License)
Outline

• 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)
    • Edges: undirected $(u,v)=(v,u)$, $u$ and $v$ adjacent; directed $(u,v)$ differs from $(v,u)$, $u$ incident to $v$;
  – Both vertices and edges may contain additional information
    • A **weight** may be associated with $(u,v)$: cost, distance, transfer function etc.

• 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
Konigsberg bridges

• Is it possible to find a route that
  – Starts and finishes at the same place?
  – Crosses each bridge exactly once?
• In 1736 Leonard Euler proved that it was not possible
  – all the vertices of the graph are odd
Some Graph Problems

- Finding shortest paths
  - Routing Internet traffic and UPS trucks
- Finding minimum spanning trees
  - Telco laying down fiber
- Identify “special” nodes and communities
  - Breaking up terrorist cells, spread of avian flu
- Bipartite matching
  - Monster.com, Match.com
- And of course... PageRank
Graphs and MapReduce

• A large class of graph algorithms 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 $n \times n$ 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 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
    • $O(n^2)$ space requirement
Adjacency Lists

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

<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
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

\[ S = \{\} ; \quad d[s] = 0 ; \quad d[v] = \text{infinity for } v \neq s \]

While \( S \neq V \)

Choose \( v \) in \( V - S \) with minimum \( d[v] \)

Add \( v \) to \( S \)

For each \( w \) in the neighborhood of \( v \)

\[ d[w] = \min(d[w], d[v] + c(v, w)) \]

Edge costs are assumed to be non-negative
Dijkstra’s Algorithm Example
Dijkstra’s Algorithm Example

Example from CLR
Dijkstra’s Algorithm Example
Dijkstra’s Algorithm Example
Dijkstra’s Algorithm Example
Dijkstra’s Algorithm Example

What happens when edge costs are negative?
Single-Source Shortest Path - MapReduce

- Single processor machine
  - Dijkstra’s Algorithm
  - Needs a priority queue that maintains globally sorted list of nodes by current distance
  - This is not possible in MapReduce
    - No means for exchanging global data

- MapReduce
  - Parallel breadth-first search (BFS)
Visualizing Parallel 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 $a$
    \[ \text{DISTANCETo}(s) = 0 \]
  - For all nodes $p$ reachable from $s$, \[ \text{DISTANCETo}(p) = 1 \]
  - For all nodes $n$ reachable from some other set of nodes $M$, \[ \text{DISTANCETo}(n) = 1 + \min(\text{DISTANCETo}(m), m \in M) \]
From Intuition to Algorithm

• Data representation:
  – Key: node $n$
  – Value: $d$ (distance from start), adjacency list (nodes reachable from $n$)
  – Initialization: for all nodes except for start node, $d = \infty$

• Mapper:
  – $\forall m \in$ adjacency list: emit $(m, d + 1)$
  – Remember to also emit distance to yourself

• 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 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 \((n, \text{adjacency list})\) as well
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
9:             d_{min} ← d ▷ Look for shorter distance
10:            M.DISTANCE ← d_{min}
11:       Emit(nid m, node M) ▷ Update shortest distance
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
- It is the diameter of the graph
  - Greatest distance between any pair of nodes
  - It is surprisingly small
  - Six degrees of separation?
  - Iterate until no more nodes have distance $\infty$
- Practicalities of implementation in MapReduce
  - See if a termination condition has been met in the “driver” program
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”
Single Source: Weighted Edges

• Now add positive weights to the edges
• 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

• 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
Additional Complexities

- Practicalities of implementation in MapReduce
  - See if a termination condition has been met in the “driver” program
Graphs and MapReduce

• A large class of graph algorithms 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
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 (query-independent)
PageRank: Defined

Given page $x$ with inlinks $t_1...t_n$, 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

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

• Then, factor in these complexities...
  – Why do we need the random jump?
  – Where do dangling nodes come from?

We only discuss till here!
Sample PageRank Iteration (1)
Sample PageRank Iteration (2)

Iteration 2

$n_1 (0.066)$

$n_2 (0.166)$

$n_3 (0.166)$

$n_4 (0.3)$

$n_5 (0.3)$

$n_2 (0.3)$

$n_3 (0.383)$

$n_4 (0.2)$
### PageRank Pseudo-Code

1. **class Mapper**
2.  
3.  
4.  
5.  
6.  

1. **class Reducer**
2.  
3.  
4.  
5.  
6.  
7.  
8.  
9.  
10.
PageRank vs. BFS

<table>
<thead>
<tr>
<th></th>
<th>PageRank</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>PR/N</td>
<td>d+1</td>
</tr>
<tr>
<td>Reduce</td>
<td>sum</td>
<td>min</td>
</tr>
</tbody>
</table>
This week in lab...

• You’ll try graph data processing with MapReduce
Next Lecture

• Joins with MapReduce and MapReduce limitations
References


• Data-Intensive Text Processing with MapReduce

Authors: Jimmy Lin and Chris Dyer

THANK YOU