Search Engine Architecture

6. Link Analysis
Today’s Agenda

- Graph problems and representations
- Parallel breadth-first search
- PageRank
- Optimizing graph algorithms
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

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

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 $j$

<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
  • 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

<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 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 \( a \)
    \[ \text{DISTANCETo}(s) = 0 \]
  • 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
Source: Wikipedia (Wave)
Via Lin et al. Big Data Infrastructure, UMD Spring 2015.
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 \((m, d + 1)\)

• 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 (\(n\), 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

Single Source: Weighted Edges

- Now add positive weights to the edges
  - Why can’t edge weights be negative?
- 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

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!

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?

Additional Complexities

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
- 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?

Sample PageRank Iteration (1)

Sample PageRank Iteration (2)

Map

Reduce

PageRank Pseudo-Code

1: class Mapper
2:   method MAP(nid n, node N)
3:     p ← N.PAGERANK/|N.ADJACENCYLIST|
4:     Emit(nid n, N)                         ▶ Pass along graph structure
5:     for all nodeid m ∈ N.ADJACENCYLIST do
6:       Emit(nid m, p)                       ▶ Pass PageRank mass to neighbors

1: class Reducer
2:   method REDUCE(nid m, [p₁, p₂, ...])
3:     M ← ∅
4:     for all p ∈ counts [p₁, p₂, ...] do
5:       if IsNode(p) then
6:         M ← p                               ▶ Recover graph structure
7:       else
8:         s ← s + p                           ▶ Sums incoming PageRank contributions
9:     M.PAGERANK ← s
10:    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

\[ p' = \alpha \left( \frac{1}{N} \right) + (1 - \alpha) \left( \frac{m}{N} + p \right) \]

• \( p \) is PageRank value from before, \( p' \) is updated PageRank value
• \( N \) is the number of nodes in the graph
• \( m \) is the missing PageRank mass
• Additional optimization: make it a single pass!

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?
  - Not a straightforward question
- Watch out for link spam:
  - Link farms
  - Spider traps
  - ...

Beyond PageRank

- Variations of PageRank
  - Weighted edges
  - Personalized PageRank
- Variants on graph random walks
  - Hubs and authorities (HITS)
  - SALSA

Applications

- Static prior for web ranking
- Identification of “special nodes” in a network
- Link recommendation
- Additional feature in any machine learning problem

Other Classes of Graph Algorithms

- Subgraph pattern matching
- Computing simple graph statistics
  - Degree vertex distributions
- Computing more complex graph statistics
  - Clustering coefficients
  - Counting triangles

Iterative Algorithms
MapReduce Sucks

- Needless graph shuffling
- Checkpointing at each iteration

MapReduce sucks at iterative algorithms

- Alternative programming models (later)
- Easy fixes (now)

In-Mapper Combining

- Use combiners
  - Perform local aggregation on map output
  - Downside: intermediate data is still materialized
- Better: in-mapper combining
  - Preserve state across multiple map calls, aggregate messages in buffer, emit buffer contents at end
  - Downside: requires memory management

Better Partitioning

• Default: hash partitioning
  • Randomly assign nodes to partitions

• Observation: many graphs exhibit local structure
  • E.g., communities in social networks
  • Better partitioning creates more opportunities for local aggregation

• Unfortunately, partitioning is hard!
  • Sometimes, chicken-and-egg...
  • But cheap heuristics sometimes available
  • For webgraphs: range partition on domain-sorted URLs

Schimmy Design Pattern

- Basic implementation contains two dataflows:
  - Messages (actual computations)
  - Graph structure (“bookkeeping”)
- Schimmy: separate the two dataflows, shuffle only the messages
- Basic idea: merge join between graph structure and messages

Schimmy Design Pattern

- Basic implementation contains two dataflows:
  - Messages (actual computations)
  - Graph structure (“bookkeeping”)
- Schimmy: separate the two dataflows, shuffle only the messages
- Basic idea: merge join between graph structure and messages

Do the Schimmy!

- Schimmy = reduce side parallel merge join between graph structure and messages
  - Consistent partitioning between input and intermediate data
  - Mappers emit only messages (actual computation)
  -Reducers read graph structure directly from HDFS

Pregel
What makes graph processing hard?

- Lessons learned so far:
  - Partition
  - Replicate
  - Reduce cross-partition communication
- What makes MapReduce “work”?

Characteristics of Graph Algorithms

• What are some common features of graph algorithms?
  • Graph traversals
  • Computations involving vertices and their neighbors
  • Passing information along graph edges

• What’s the obvious idea?
  • Keep “neighborhoods” together!

Simple Partitioning Techniques

- Hash partitioning
- Range partitioning on some underlying linearization
  - Web pages: lexicographic sort of domain-reversed URLs
  - Social networks: sort by demographic characteristics

Country Structure in Facebook

Ugander et al. (2011) The Anatomy of the Facebook Social Graph.

Analysis of 721 million active users (May 2011)

54 countries w/ >1m active users, >50% penetration
What makes graph processing hard?

• It’s tough to apply our “usual tricks”:
  • Partition
  • Replicate
  • Reduce cross-partition communication

Pregel: Computational Model

- Based on Bulk Synchronous Parallel (BSP)
  - Computational units encoded in a directed graph
  - Computation proceeds in a series of supersteps
  - Message passing architecture
- Each vertex, at each superstep:
  - Receives messages directed at it from previous superstep
  - Executes a user-defined function (modifying state)
  - Emits messages to other vertices (for the next superstep)
- Termination:
  - A vertex can choose to deactivate itself
  - Is “woken up” if new messages received
  - Computation halts when all vertices are inactive

Pregel

superstep $t$

superstep $t+1$

superstep $t+2$

Pregel: Implementation

- Master-Slave architecture
  - Vertices are hash partitioned (by default) and assigned to workers
  - Everything happens in memory
- Processing cycle:
  - Master tells all workers to advance a single superstep
  - Worker delivers messages from previous superstep, executing vertex computation
  - Messages sent asynchronously (in batches)
  - Worker notifies master of number of active vertices
- Fault tolerance
  - Checkpointing
  - Heartbeat/revert

Pregel: PageRank

class PageRankVertex : public Vertex<double, void, double> {
public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
                sum += msgs->Value();
            *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
        }

        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue() / n);
        } else {
            VoteToHalt();
        }
    }
};

Questions?