Cloud Computing Lecture #5

Graph Algorithms with MapReduce

Jimmy Lin
The iSchool
University of Maryland

Wednesday, October 1, 2008

Today's Topics

- Introduction to graph algorithms and graph representations
- Single Source Shortest Path (SSSP) problem
  - Refresher: Dijkstra's algorithm
  - Breadth-First Search with MapReduce
- PageRank

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

- Graph algorithms typically involve:
  - Performing computation at each node
  - Processing node-specific data, edge-specific data, and link structure
  - Traversing the graph in some manner
- Key questions:
  - How do you represent graph data in MapReduce?
  - How do you traverse a graph in MapReduce?

Representing Graphs

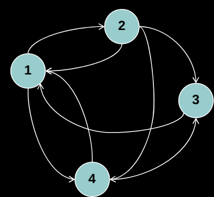
- G = (V, E)
  - A poor representation for computational purposes
- Two common representations
  - Adjacency matrix
  - Adjacency list

Adjacency Matrices

Represent a graph as an *n* x *n* square matrix *M*

- $n = |V|$
- $M_{ij} = 1$ means a link from node *i* to *j*

	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0



Adjacency Matrices: Critique

- Advantages:
  - Naturally encapsulates iteration over nodes
  - Rows and columns correspond to inlinks and outlinks
- Disadvantages:
  - Lots of zeros for sparse matrices
  - Lots of wasted space

The iSchool
University of Maryland

Adjacency Lists

Take adjacency matrices… and throw away all the zeros

	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0

→

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

The iSchool
University of Maryland

Adjacency Lists: Critique

- Advantages:
  - Much more compact representation
  - Easy to compute over outlinks
  - Graph structure can be broken up and distributed
- Disadvantages:
  - Much more difficult to compute over inlinks

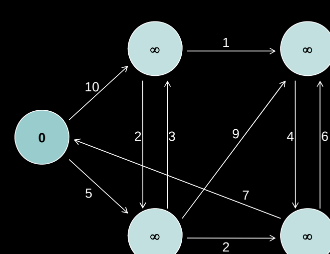
The iSchool
University of Maryland

Single Source Shortest Path

- **Problem:** find shortest path from a source node to one or more target nodes
- First, a refresher: Dijkstra's Algorithm

The iSchool
University of Maryland

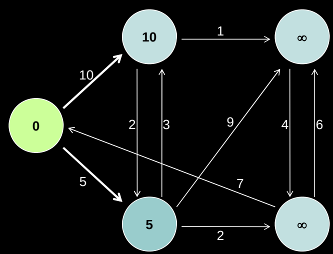
Dijkstra's Algorithm Example



Example from CLR

The iSchool
University of Maryland

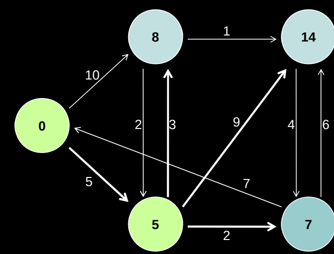
Dijkstra's Algorithm Example



Example from CLR

The iSchool
University of Maryland

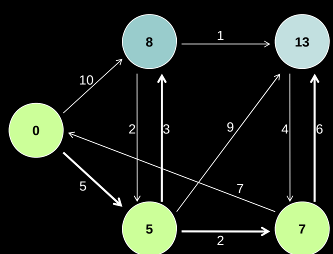
Dijkstra's Algorithm Example



Example from CLR

The iSchool
University of Maryland

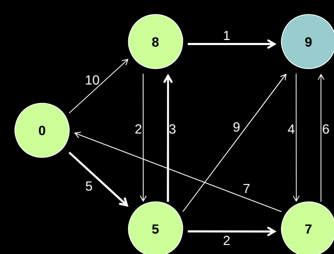
Dijkstra's Algorithm Example



Example from CLR

The iSchool
University of Maryland

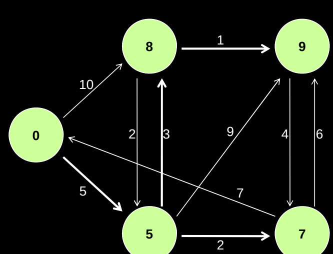
Dijkstra's Algorithm Example



Example from CLR

The iSchool
University of Maryland

Dijkstra's Algorithm Example



Example from CLR

The iSchool
University of Maryland

Single Source Shortest Path

- **Problem:** find shortest path from a source node to one or more target nodes
- Single processor machine: Dijkstra's Algorithm
- MapReduce: parallel Breadth-First Search (BFS)

The iSchool
University of Maryland

Finding the Shortest Path

- First, consider equal edge weights
- Solution to the problem can be defined inductively
- Here's the intuition:
  - 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 $\in$ S)

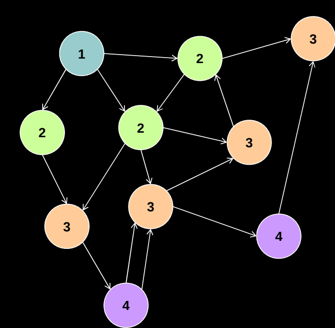
From Intuition to Algorithm

- A map task receives
  - Key: node *n*
  - Value: D (distance from start), points-to (list of nodes reachable from *n*)
- $\forall p \in$ points-to: emit (*p*, D+1)
- The reduce task gathers possible distances to a given *p* and selects the minimum one

Multiple Iterations Needed

- This MapReduce task advances the "known frontier" by one hop
  - Subsequent iterations include more reachable nodes as frontier advances
  - Multiple iterations are needed to explore entire graph
  - Feed output back into the same MapReduce task
- Preserving graph structure:
  - Problem: Where did the points-to list go?
  - Solution: Mapper emits (*n*, points-to) as well

Visualizing Parallel BFS



Termination

- Does the algorithm ever terminate?
  - Eventually, all nodes will be discovered, all edges will be considered (in a connected graph)
- When do we stop?

Weighted Edges

- Now add positive weights to the edges
- Simple change: points-to list in map task includes a weight *w* for each pointed-to node
  - emit (*p*, D+w_p) instead of (*p*, D+1) for each node *p*
- Does this ever terminate?
  - Yes! Eventually, no better distances will be found. When distance is the same, we stop
  - Mapper should emit (*n*, D) to ensure that "current distance" is carried into the reducer

Comparison to Dijkstra

- Dijkstra's algorithm is more efficient
  - At any step it only pursues edges from the minimum-cost path inside the frontier
- MapReduce explores all paths in parallel
  - Divide and conquer
  - Throw more hardware at the problem

General Approach

- MapReduce is adapt at manipulating graphs
  - Store graphs as adjacency lists
- Graph algorithms with for MapReduce:
  - Each map task receives a node and its outlinks
  - Map task compute some function of the link structure, emits value with target as the key
  - Reduce task collects keys (target nodes) and aggregates
- Iterate multiple MapReduce cycles until some termination condition
  - Remember to "pass" graph structure from one iteration to next

The iSchool
University of Maryland

Random Walks Over the Web

- Model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- What's the amount of time that will be spent on any given page?
- This is PageRank

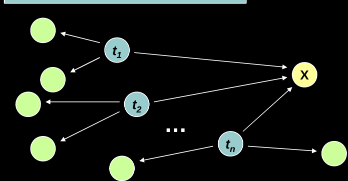
The iSchool
University of Maryland

PageRank: Defined

Given page x with in-bound links $t_1 \ldots t_n$, where

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

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



The iSchool
University of Maryland

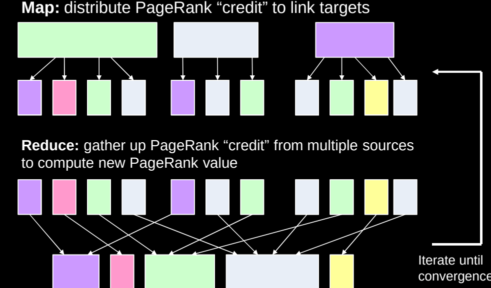
Computing PageRank

- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration is 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

The iSchool
University of Maryland

PageRank in MapReduce

Map: distribute PageRank "credit" to link targets



The iSchool
University of Maryland

PageRank: Issues

- Is PageRank guaranteed to converge? How quickly?
- What is the “correct” value of α, and how sensitive is the algorithm to it?
- What about dangling links?
- How do you know when to stop?

The iSchool
University of Maryland

Questions?

(Ask them now, because you’re going to have to implement this!)