Graph Algorithms

Revised based on the slides by Ruoming Jin @ Kent State

1

1

Outlines

- Graph problems and representations
- Parallel breadth-first search
- PageRank

2

2

Outlines

- Graph problems and representations
- Parallel breadth-first search
- PageRank

3

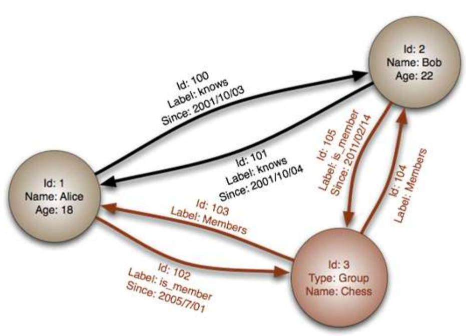
3

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

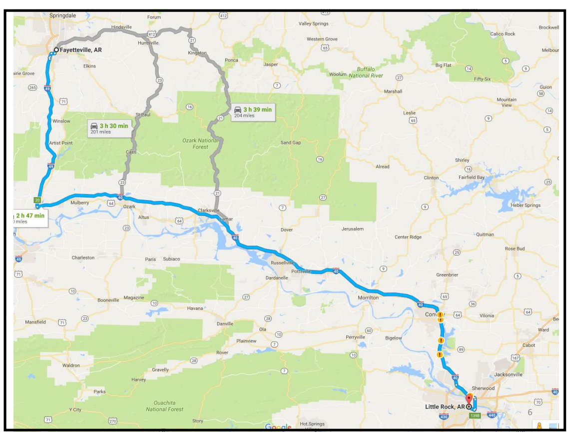
4

4



5

5



6

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

7

7

Graphs and MapReduce

- Graph algorithms typically 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?

8

8

Representing Graphs

- G = (V, E)
- Two common representations
  - Adjacency matrix
  - Adjacency list

9

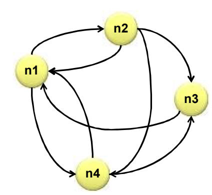
9

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

	n1	n2	n3	n4
n1	0	1	0	1
n2	1	0	1	1
n3	1	0	0	0
n4	1	0	1	0



10

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

11

11

Adjacency Lists

Take adjacency matrices… and throw away all the zeros

	n1	n2	n3	n4
n1	0	1	0	1
n2	1	0	1	1
n3	1	0	0	0
n4	1	0	1	0

→

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

12

Adjacency Lists: Critique

- Advantages:
  – Much more compact representation
  – Easy to compute over outlinks
- Disadvantages:
  – Much more difficult to compute over inlinks

13

13

Outlines

- Graph problems and representations
- Parallel breadth-first search
- PageRank

14

14

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

15

15

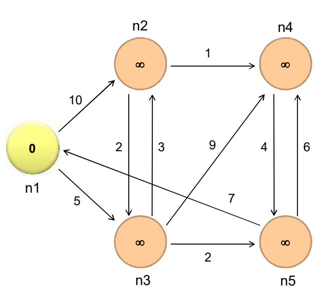
Dijkstra's Algorithm

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*. Set the initial node as the current node.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance with the current assigned value, and assign the smaller one.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If all of destination nodes have been marked visited or if the smallest tentative distance among the nodes in the *unvisited set* is infinity, then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

16

16

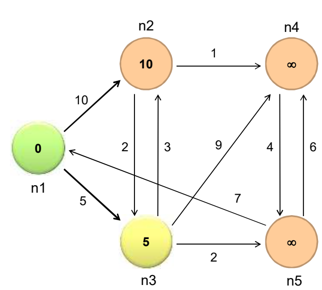
Dijkstra's Algorithm Example



Example from CLR

17

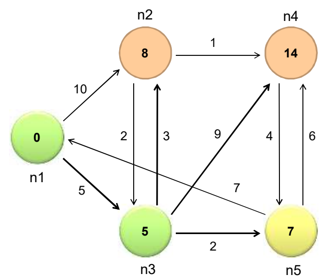
Dijkstra's Algorithm Example



Example from CLR

18

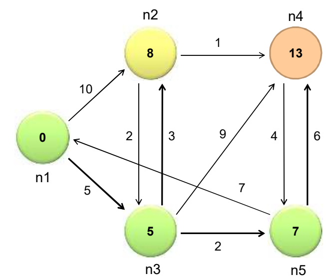
Dijkstra's Algorithm Example



Example from CLR

19

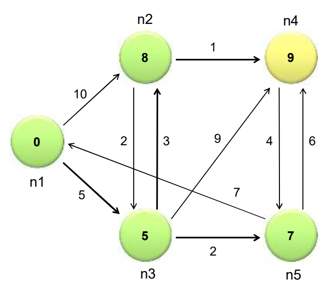
Dijkstra's Algorithm Example



Example from CLR

20

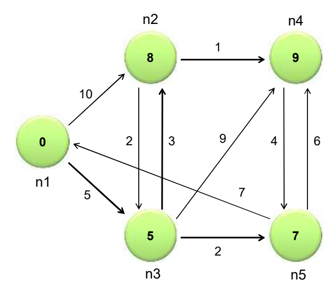
Dijkstra's Algorithm Example



Example from CLR

21

Dijkstra's Algorithm Example



Example from CLR

22

Dijkstra's Algorithm

$\textsc{Dijkstra}(G, w, s)$

  $\cancel{d[s] \leftarrow 0}$

  for all vertex $v \in V$ **do**

    $d[v] \leftarrow \infty$

  $d[s] \leftarrow 0$

  $Q \leftarrow \{V\}$

  while $Q \neq \emptyset$ **do**

    $u \leftarrow \textsc{ExtractMin}(Q)$

    for all vertex $v \in u.\textsc{AdjacencyList}$ **do**

      if $d[v] > d[u] + w(u, v)$ **then**

        $d[v] \leftarrow d[u] + w(u, v)$

23

Single Source Shortest Path

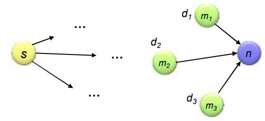
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Single processor machine: Dijkstra's Algorithm
- MapReduce: parallel Breadth-First Search (BFS)

24

24

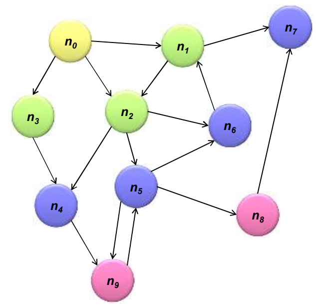
Finding the Shortest Path

- Consider simple case of equal edge weights (i.e., weight=1)
- 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*
  - DISTANCETO(*s*) = 0
  - For all nodes *p* reachable from *s*, DISTANCETO(*p*) = 1
  - For all nodes *n* reachable from some other set of nodes *M*, DISTANCETO(*n*) = 1 + min(DISTANCETO(*m*), $m \in M$)



25

Visualizing Parallel BFS



26

From Intuition to Algorithm

- Data representation:
  - Key: node *n*
  - Value: *d* (distance from start), adjacency list (list of 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

27

27

Multiple Iterations Needed

- Each MapReduce iteration advances the "known 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

28

28

BFS Pseudo-Code

```
class Mapper
    method Map(nid n, node N)
        d ← N.Distance
        Emit(nid n, N)                          ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, d + 1)                  ▷ Emit distances to reachable nodes

class Reducer
    method Reduce(nid m, [d1, d2, . . .])
        dmin ← ∞
        M ← ∅
        for all d ∈ counts [d1, d2, . . .] do
            if IsNode(d) then
                M ← d                           ▷ Recover graph structure
            else if d < dmin then               ▷ Look for shorter distance
                dmin ← d
        M.Distance ← dmin                       ▷ Update shortest distance
        Emit(nid m, node M)
```

if dmin < current distance, update; otherwise, keep the current distance

29

29

Stopping Criterion

- How many iterations are needed in parallel BFS (equal edge weight case)?
  - Six degrees of separation?
- Practicalities of implementation in MapReduce

30

30

Comparison with 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
  - Lots of "waste"
  - Useful work is only done at the "frontier"
    - Non-useful work can be avoided

31

31

Weighted Edges

- Now add positive weights to the edges
- Simple change: adjacency list now includes a weight *w* for each edge
  - In mapper, emit (m, $d + w_p$) instead of (m, $d + 1$) for each node m

32

32

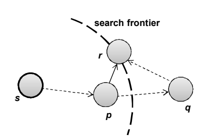
Stopping Criterion

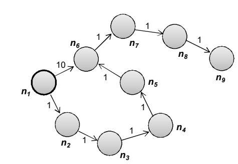
- How many iterations are needed in parallel BFS (positive edge weight case)?
- Convince yourself: when a node is first "discovered", we've found the shortest path
  Not true!
  - A node becomes "discovered" when the cost of the node becomes non-infinity.

33

33

Additional Complexities





34

Graphs and MapReduce

- Graph algorithms typically 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

35

35

A practical implementation

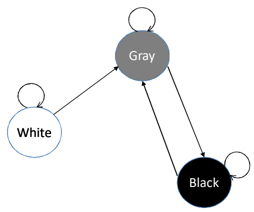
- Referenced from the following link
  - http://www.johnandcailin.com/blog/cailin/breadth-first-graph-search-using-iterative-map-reduce-algorithm
- A node is represented by a string as follows
  - ID EDGES|WEIGHTS|DISTANCE_FROM_SOURCE|COLOR

36

36

Three statuses of a node

- Unvisited
  - Color white
- Being visited
  - Color gray
- Visited
  - Color black



37

37

The mappers

- All white nodes and black nodes only reproduce themselves
- For each gray node (e.g., an exploding node)
  - For each node n in the adjacency list, emit a gray node
    - n null|null|distance of exploding node + weight|gray
  - Turn its own color to black and emit itself
    - ID edges|weights|distance from source|black

38

38

The reducers

- Receive the data for all “copies” of each node
- Construct a new node for each node
  - The non-null list of edges and weights
  - The minimum distance from the source
  - The proper color

39

39

Choose the proper color

- If only receiving a copy of white node, color is white
- If only receiving a copy of black node, color is black
- If receiving copies consisting of white node and gray nodes, color is gray
- If receiving copies consisting of gray nodes and black node
  - If minimum distance comes from black node, color is black
  - Otherwise, color is gray

40

40

Outlines

- Graph problems and representations
- Parallel breadth-first search
- PageRank

41

41

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
  - Or, sometimes, user jumps to a random page
- PageRank
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages
- PageRank captures notions of page importance
  - One of thousands of features used in web search

42

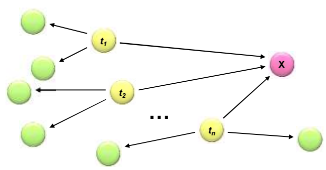
42

PageRank: Defined

Given page x with inlinks $t_1 \ldots t_n$, where

- $C(t_i)$ is the out-degree of t_i, i.e., the number of outgoing links from t_i
- α is probability of random jump
- N is the total number of nodes in the graph

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



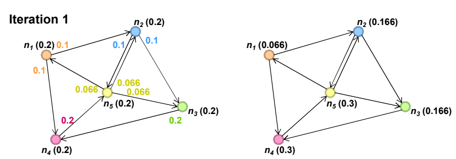
43

Computing PageRank

- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm (ignoring random jump):
  - Start with seed PR_i values
  - Each page distributes PR_i mass to all pages it links
  - Each target page adds up mass from multiple in-bound links to compute PR_{i+1}
  - Iterate until values converge

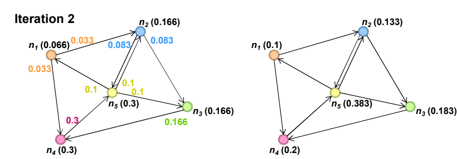
44

Sample PageRank Iteration (1)



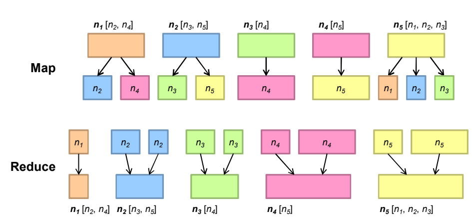
45

Sample PageRank Iteration (2)



46

PageRank in MapReduce



47

PageRank Pseudo-Code

```
class Mapper
    method Map(nid n, node N)
        p ← N.PageRank/|N.AdjacencyList|
        Emit(nid n, N)                              ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, p)                          ▷ Pass PageRank mass to neighbors

class Reducer
    method Reduce(nid m, [p1, p2, ...])
        M ← ∅, s = 0
        for all p ∈ counts [p1, p2, ...] do
            if IsNode(p) then
                M ← p                               ▷ Recover graph structure
            else
                s ← s + p                           ▷ Sum incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```

48

Complete PageRank

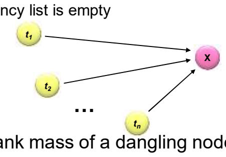
- Two additional complexities
  - What is the proper treatment of dangling nodes?
  - How do we factor in the random jump factor?

49

49

Dangling nodes

- A dangling node is a node that has no outgoing edges
  - The adjacency list is empty



- The PageRank mass of a dangling node will get lost during the mapper stage due to the lack of outgoing edges
- Solution
  - Reserve a special key (i.e., a special node id) for storing PageRank mass from dangling nodes
    - Mapper: dangling nodes emit the mass with the special key
    - Reducer: sum up all the missing mass with the special key

50

Second pass

- Second pass to redistribute "missing PageRank mass" and account for random jumps

$$p' = \alpha\left(\frac{1}{|G|}\right) + (1-\alpha)\left(\frac{m}{|G|} + p\right)$$

  - p is PageRank value from the first pass, p' is updated PageRank value
  - |G| is the number of nodes in the graph
  - m is the combined missing PageRank mass

51

51

Complete PageRank

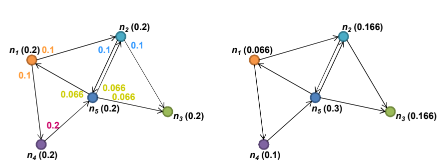
- One iteration of PageRank requires two passes (i.e., two MapReduce jobs)
  - The first to distribute PageRank mass along graph edges
    - Also take care of the missing mass due to dangling nodes
  - The second to redistribute the missing mass and take into account the random jump factor
    - This job requires no reducers

52

52

Sample PageRank Iteration (1)

Iteration 1
Pass 1



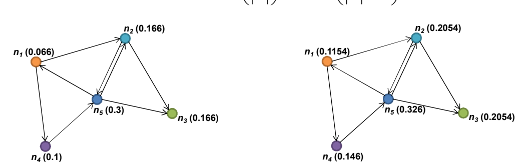
Missing PR mass = 0.2

53

Sample PageRank Iteration (1)

Iteration 1
Pass 2

$$p' = \alpha\left(\frac{1}{|G|}\right) + (1-\alpha)\left(\frac{m}{|G|} + p\right)$$



Missing PR mass = 0.2

α = 0.1, m = 0.2

54

PageRank Convergence

- Convergence criteria
  - Iterate until PageRank values don't change
  - Fixed number of iterations
- Convergence for web graphs?
  - 52 iterations for a graph with 322 million edges

55

55