Data Mining and Networks

- Data mining has rich history and methods for analyzing. . .
  - . . . tabular data
  - . . . textual data
  - . . . time series and streams
  - . . . market baskets

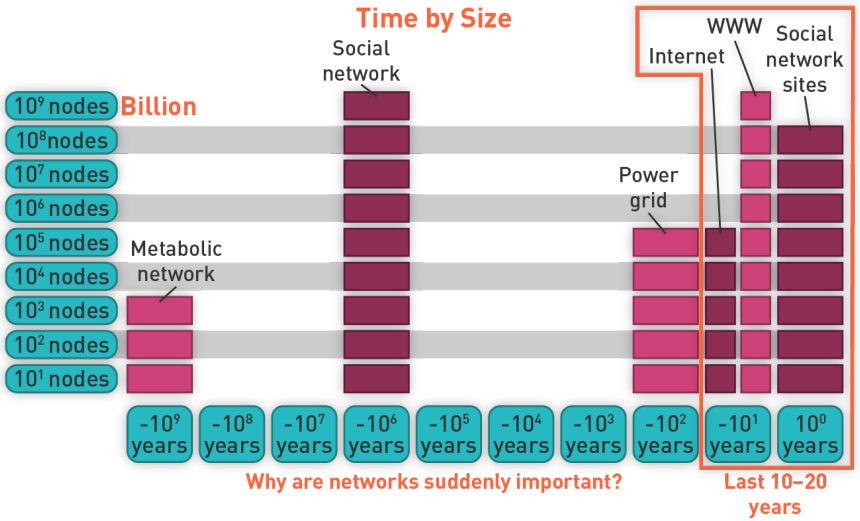
Bag of features

- **What about relations and dependencies?**

Networks Are Useful

- Networks allow for modeling dependencies.
- Networks are a general language for describing real-world systems.
- Only recently have they become a first-class citizen of the machine-learning world.

The Life of Networks



Networks Are Transforming Society

- Networks are transforming how we socialize, how we learn, how we exercise, how we communicate...and how we compute.
- Networks are much more sophisticated today than 10 years ago...
- Provide a lot of insight into nature and humanity and overall have high utility.
- How we work, rest, and play but also how we compute.

Mining Network Data

Opens up great new worlds of solutions applications, *but* it also opens many challenges...

- Working with network data is messy.
  - Not just "wiring diagrams" but also dynamics and data (features, attributes) on nodes and edges
- Computational challenges.
  - Large-scale network data
- Algorithmic models as vocabulary for expressing complex scientific questions.
  - Social science, physics, biology

AT&T

- Local mobile search
- Social data
- Struggled to compute
- Neo4j

Networks and Graphs Are a First-Class Citizen

- Networks rich source of problems and applications.
- Great source of features.
- Also a great way of solving a variety of machine-learning problems.
    - Bayesian networks
    - Bipartite graph problems such as recommendation engines
- Synergies between graph processing and machine are manyfold.
    - Hybrid of graph traversal (random walks) + classification
    - Filling in missing data in networked data

Networks Mathematically

- Represent networks mathematically as graphs of nodes and edges.
- This lecture focuses on graph algorithms in MapReduce.

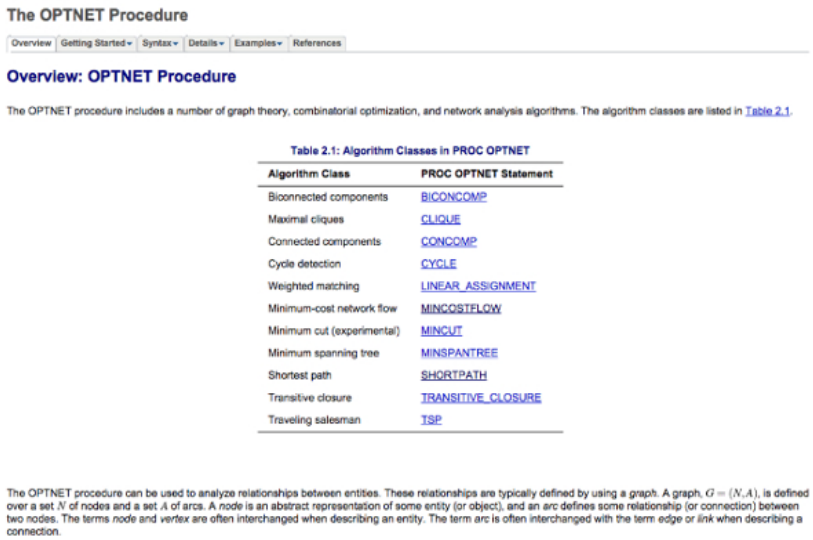
Some Graph Problems

- Graph search and path planning
  - Routing Internet traffic and UPS delivery trucks
  - Finding experts, friend recommendations
- Finding minimum spanning trees
  - Telco laying down fiber
- Finding max flow
  - Airline scheduling

Some Graph Problems (cont.)

- Identify "special" nodes and communities.
  - Breaking up terrorist cells, spread of avian flu
- Bipartite matching.
  - Monster.com (employer-applicant), Match.com (person-person), Netflix (viewer-movie), recommender systems, finding restaurants
- Text summarization, concept extraction (TextRank).
- Find popular web pages: PageRank, Hits.

http://support.sas.com/



The OPTNET Procedure

Overview | Getting Started | Syntax | Details | Examples | References

Overview: OPTNET Procedure

The OPTNET procedure includes a number of graph theory, combinatorial optimization, and network analysis algorithms. The algorithm classes are listed in Table 2.1.

Table 2.1: Algorithm Classes in PROC OPTNET

Algorithm Class	PROC OPTNET Statement
Biconnected components	BICONCOMP
Maximal cliques	CLIQUE
Connected components	CONCOMP
Cycle detection	CYCLE
Weighted matching	LINEAR_ASSIGNMENT
Minimum-cost network flow	MINCOSTFLOW
Minimum cut (experimental)	MINCUT
Minimum spanning tree	MINSPANTREE
Shortest path	SHORTPATH
Transitive closure	TRANSITIVE_CLOSURE
Traveling salesman	TSP

The OPTNET procedure can be used to analyze relationships between entities. These relationships are typically defined by using a *graph*. A graph, $G = (N, A)$, is defined over a set N of nodes and a set A of arcs. A *node* is an abstract representation of some entity (or object), and an *arc* defines some relationship (or connection) between two nodes. The terms *node* and *vertex* are often interchanged when describing an entity. The term *arc* is often interchanged with the term *edge* or *link* when describing a connection.

Single Machine? MapReduce?

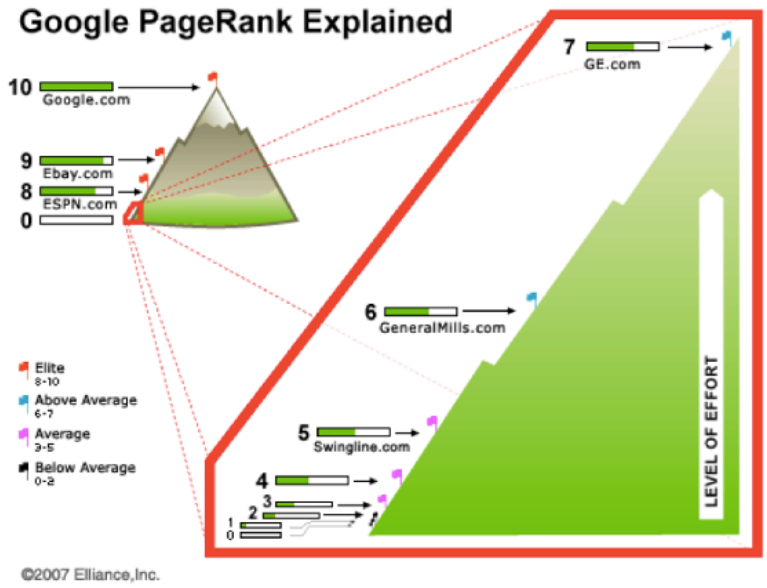
- A common feature of these problems is the scale of the datasets on which the algorithms must operate, for example, the hyperlink structure of the web, which contains billions of pages, or social networks that contain hundreds of millions of individuals. Clearly, algorithms that run on a single machine and depend on the entire graph residing in memory are not scalable. We'd like to put MapReduce to work on these challenges.
- E.g., Dijkstra's algorithm. However, this famous algorithm assumes sequential processing.

Figure 2.3: Shortest Path for Road Network at 5:00 P.M.

order	start_inter	end_inter	time_to_travel
1	SASCampusDrive	US40W/HarrisonAve	1.2000
2	US40W/HarrisonAve	RaleighExpy/US40W	1.4182
3	RaleighExpy/US40W	US440W/RaleighExpy	3.0000
4	US440W/RaleighExpy	US70W/US440W	2.7000
5	US70W/US440W	Capital/US70W	3.2000
6	Capital/US70W	614CapitalBlvd	1.4400
			12.9582

- Seasonality
  - Rush hour
  - Game night

Popularity on the Web: PageRank



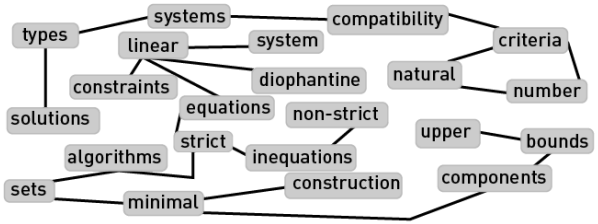
Finding Friends

Linking other things such as groups

- Growing body of research captures dynamics of social network graphs.
  - [Latanzi, Sivakumar '08][Zheleva, Sharara, Getoor '09] [Kumar, Novak, Tomkins '06][Kossinets, Watts '06][L., Kleinberg, Faloutsos '05]
- What links will occur next? [LibenNowell, Kleinberg '03]
  - Networks + many other features:
    - Location, school, job, hobbies, interests, etc.

TextRank: An Example

Compatibility of systems of linear constraints over the set of natural numbers Criteria of compatibility of a system of linear Diophantine equations, strict inequations, and nonstrict inequations are considered. Upper bounds for components of a minimal set of solutions and algorithms of construction of minimal generation sets of solutions for all types of systems are given. These criteria and the corresponding algorithms for constructing a minimal supporting set of solutions can be used in solving all the considered types of systems and systems of mixed types.



TextRank

numbers (1.46)
inequations (1.45)
linear (1.29)
diophantine (1.28)
upper (0.99)
bounds (0.99)
strict (0.77)

Frequency

systems (4)
types (4)
solutions (3)
minimal (3)
linear (2)
inequations (2)
algorithms (2)

Keywords by TextRank: *linear constrains, linear diophantine equations natural numbers, non-strict inequations, strict inequations, upper bounds*
Keywords by human annotators: *linear constraints, linear diophantine equations, non-strict inequations, set of natural numbers, strict inequations, upper bounds*

~ 100% match

Lecture Outline

The focus of this lecture:

- Graph-based algorithms
  - Shortest path
  - Single-source shortest path (SSSP)
  - Review Dijkstra's algorithm (on single machine)
  - Distributed SSSP

The focus of later lectures:

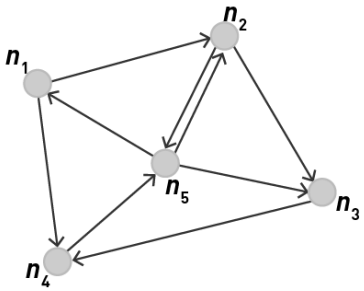
- PCA and Graphs
  - PCA
  - PageRank
- Social networks
  - Friends-of-friends (FoF): how to help expand the interconnectedness of a network
- Combine TF_IDF with page rank

Graphs (cont.)

- Different types of graphs:
  - Directed vs. undirected edges
  - Presence or absence of cycles

Graph Representations

- Graph G = (V,E)
  - V represents the set of vertices (nodes).
  - E represents the set of edges (links).
  - Both vertices and edges may contain additional information.
- E.g.
  - V={1,2,3,4,5}
  - E={(1,2)(1,5)(2,5)(2,4)(4,5)(2,3)(2,4)}
- Standard ways to represent a graph
  - Vertices and edges: G = (V, E): Not food for computational purposes



	n_1	n_2	n_3	n_4	n_5
n_1	0	1	0		0
n_2	0	0	1	0	1
n_3	0	0	0	1	0
n_4	0	0	0	0	1
n_5	1	1	1	0	0

adjacency matrix

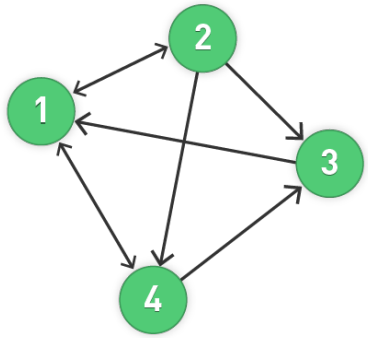
n_1 $[n_2, n_4]$
n_2 $[n_2, n_4]$
n_3 $[n_2, n_4]$
n_4 $[n_2, n_4]$
n_5 $[n_1, n_2, n_3]$

adjacency lists

Adjacency Matrices

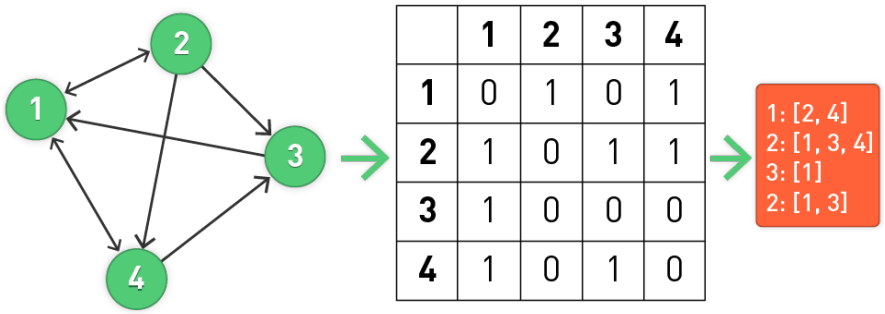
- Represent a graph as an n × n square matrix M.
  - n = |V|
  - Mij = 1 means a link from node i to j
- Advantages:
  - Naturally encapsulates iteration over nodes.
  - Rows and columns correspond to inlinks and outlinks.
- Disadvantages:
  - Lots of zeros for sparse matrices
  - Lots of wasted space
- Standard ways to represent a graph
  - Vertices and edges: G = (V, E): not good for computational purposes

	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 Lists

- An array Adj of |V| lists.
- For each u in V, the adjacency list Adj[u] contains all the vertices v such that (u,v) in E.
- 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

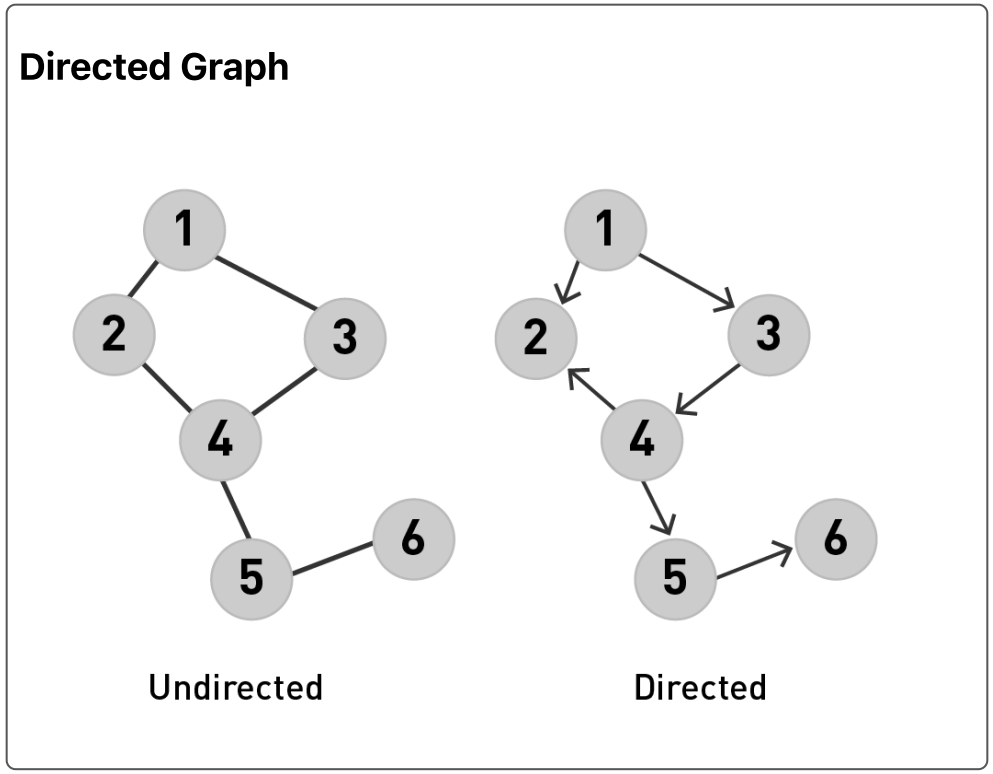
1: [2, 4]
2: [1, 3, 4]
3: [1]
2: [1, 3]

Adjacency List Is Preferred

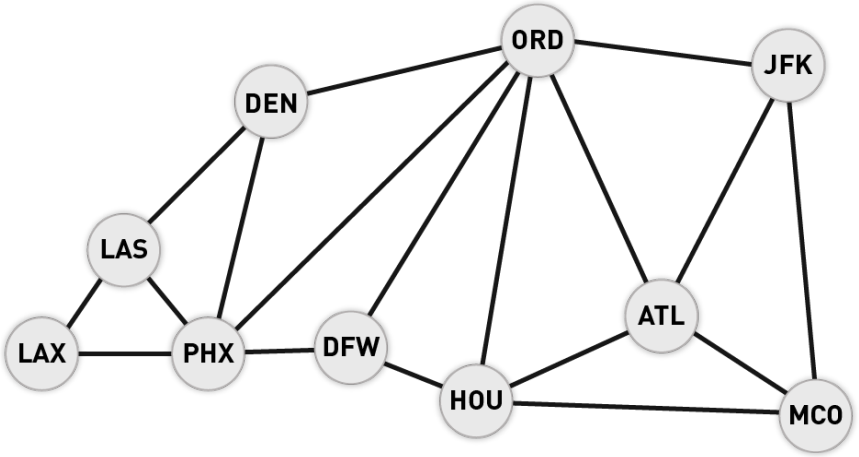
- Adjacency list is usually preferred, because it provides a compact way to represent **sparse** graphs: those for which |E| is much less than |V|.
- Adjacency matrix may be preferred when the graph is **dense**, or when we need to be able to tell quickly if there is an edge connecting two given vertices.
- Adjacency list is preferred in the following:
  - In a social network of n individuals, there are n(n – 1) possible friendships (where n may be on the order of billions). However, even the most gregarious will have relatively few friends compared to the size of the network (thousands, perhaps, but still far smaller than hundreds of millions).
  - The same is true for the hyperlink structure of the web: Each individual web page links to a minuscule portion of all the pages on the web.

Paths and Graph Traversals

- In graph theory, a path in a graph is a finite or infinite sequence of edges that connect a sequence of vertices, which, by most definitions, are all distinct from one another.
- Directed path:
    - In a directed graph, a directed path (sometimes called dipath) is again a sequence of edges (or arcs) that connect a sequence of vertices but with the added restriction that the edges all be directed in the same direction.


Directed Graph
1
2
3
4
5
6
Undirected
1
2
3
4
5
6
Directed

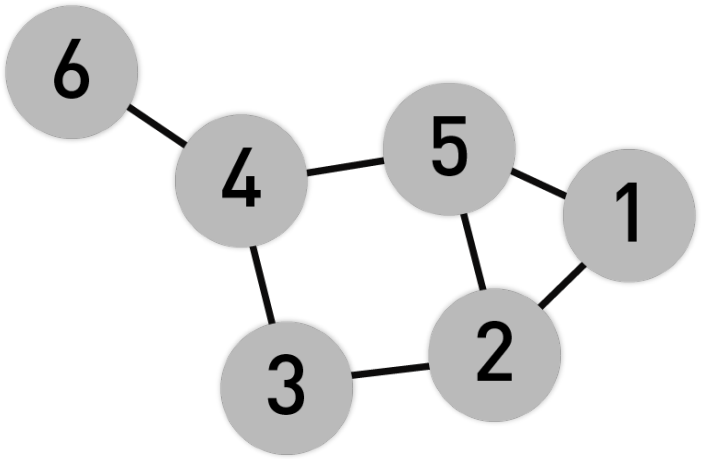
Graph Traversal: A Path as an Alternating Sequence of Vertices and Edges



Source	Destination	Distance	Shortest Paths
JFK	LAX	3	JFK-ORD-PHX-LAX
LAS	MCO	4	LAS-PHX-DFW-HOU-MCO and four others
HOU	JFK	2	HOU-ATL-JFK and two others

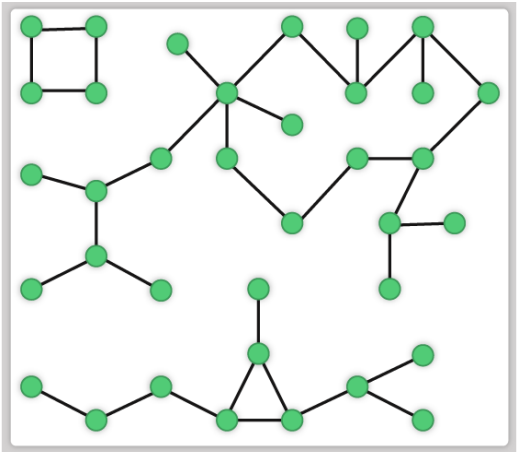
A path/walk of length k in a graph is an alternating sequence of vertices and edges, v_0,e_0,v_1,e_1,v_2,..., v_{k-1},e_{k-1},v_k, which begins and ends with vertices.

Graph Properties



An example graph, with the properties of being **planar** and being **connected**, and with order 6, size 7, **diameter** 3, **girth** 3, **vertex connectivity** 1, and **degree sequence** <3, 3, 3, 2, 2, 1>.

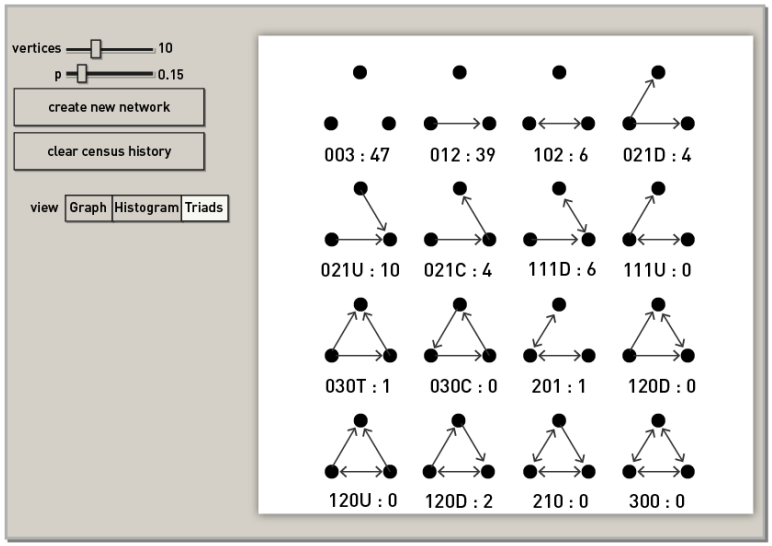
Connected Component



A graph with three connected components.

A connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

Study Graphs Using Triads



Graph Algorithms

Pages in category "Graph algorithms"

The following 103 pages are in this category, out of 103 total. This list may not reflect recent changes (learn more).

A

- A* search algorithm
- Algorithmic version for Szemerédi regularity partition
- Alpha–beta pruning
- Aperiodic graph

B

- B*
- Barabási–Albert model
- Belief propagation
- Bellman–Ford algorithm
- Bianconi–Barabási model
- Bidirectional search
- Borůvka's algorithm
- Bottleneck traveling salesman problem
- Breadth-first search
- Bron–Kerbosch algorithm

C

- Centrality
- Chaitin's algorithm
- Christofides algorithm
- Clique percolation method
- Euler tour technique

F

- FKT algorithm
- Flooding algorithm
- Flow network
- Floyd–Warshall algorithm
- Force-directed graph drawing
- Ford–Fulkerson algorithm
- Fringe search

G

- Girvan–Newman algorithm
- Goal node (computer science)
- Gomory–Hu tree
- Graph bandwidth
- Graph embedding
- Graph isomorphism
- Graph isomorphism problem
- Graph kernel
- Graph reduction
- Graph traversal

H

- Hierarchical closeness
- Lexicographic breadth-first search
- Longest path problem

M

- Minimax
- Minimum bottleneck spanning tree
- Minimum cut
- Misra & Gries edge coloring algorithm

N

- Nearest neighbour algorithm
- Network simplex algorithm
- Nonblocking minimal spanning switch

P

- Path-based strong component algorithm
- Prim's algorithm
- Proof-number search
- Push–relabel maximum flow algorithm

R

- Reverse-delete algorithm
- Rocha-Thatte cycle detection algorithm

S

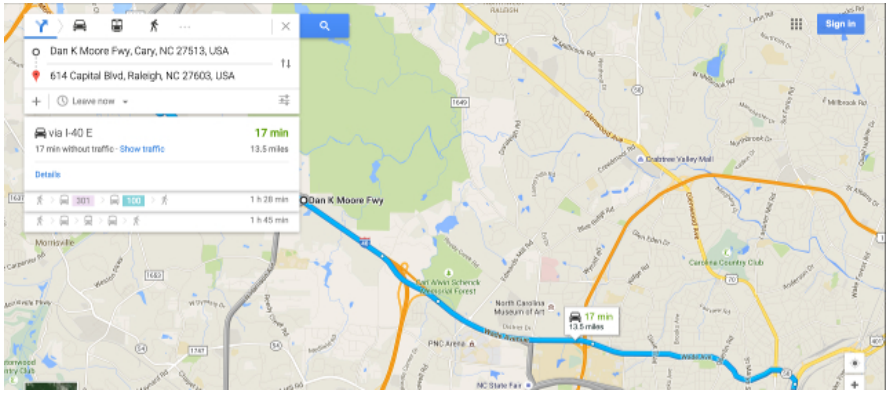
http://en.wikipedia.org/wiki/List_of_algorithms#Graph_algorithms

Introduction

- Previously we talked a bit about graphs: how to represent them and how to traverse them.
- In this section we will discuss one of the most important graph algorithms: Dijkstra's shortest-path algorithm, a greedy algorithm that efficiently finds shortest paths in a graph.

Figure 2.3: Shortest Path for Road Network at 5:00 P.M.

order	start_inter	end_inter	time_to_travel
1	SASCampusDrive	US40W/HarrisonAve	1.2000
2	US40W/HarrisonAve	RaleighExpy/US40W	1.4182
3	RaleighExpy/US40W	US440W/RaleighExpy	3.0000
4	US440W/RaleighExpy	US70W/US440W	2.7000
5	US70W/US440W	Capital/US70W	3.2000
6	Capital/US70W	614CapitalBlvd	1.4400
			12.9582



Finding the Shortest Path

- Commonly done on a single machine with Dijkstra's algorithm.
- Can we use BFS to find the shortest path via MapReduce?
- A common graph-search application is finding the shortest path from a start node to one or more target nodes.
- This is called the single-source shortest-path problem (aka SSSP).

Single Source: Smallest Number of Edges That Must Be Traversed in Order to Get to Every Vertex

- From a single source reach all other nodes in shortest way possible (unweighted graph).
  - Let's consider a simpler problem: solving the single-source shortest-path problem for an unweighted directed graph.
- In this case we are trying to find the smallest number of edges that must be traversed in order to get to every vertex in the graph.
- This is the same problem as solving the weighted version where all the weights happen to be 1.
- Sometimes we want to reach all nodes in a graph from a single starting node. How can we do this in the shortest way possible?

Single Source: Smallest Number of Edges That Must Be Traversed in Order to Get to Every Vertex

- From a single source reach all other nodes in shortest way possible (unweighted graph).
    - Let's consider a simpler problem: solving the single-source shortest-path problem for an unweighted directed graph.
- Sometimes we want to reach all nodes in a graph from a single starting node. How can we do this in the shortest way possible?
- Sometimes we just have one target destination in mind.
- We will look at algorithms for this over the next couple of sections.

Single-Source Shortest-Path Unweighted Graphs

- Single source: smallest number of edges that must be traversed in order to get to every vertex
- Unweighted graphs

Single Source: Smallest Number of Edges That Must Be Traversed in Order to Get to Every Vertex

- Challenge 1: Shortest path to all nodes from a source
  - Sometimes we want to reach all nodes in a graph from a single starting node. How can we do this in the shortest way possible?
- Challenge 2: Shortest path to a single destination
  - Sometimes we just have one target destination in mind.
- Do we know an algorithm for determining this?
  - !! Yes: breadth-first search.

Overview

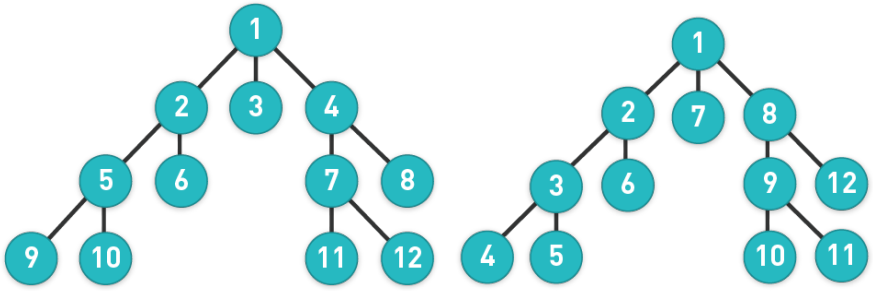
- Since this is our first graph processing algorithm in this course, we are going to spend time on this.
- Drill into the details.
- If at any time in this section you feel you have mastered BFS/DFS for unweighted graphs, please feel free to skip to the end of this section.
- Otherwise, enjoy the tour and the animations!

Single Source: Smallest Number of Edges That Must Be Traversed in Order to Get to Every Vertex

- Shortest is relative.
  - Unweighted
  - Weighted (sum of edge weights on path from source node to target node)
- Graph traversal.
  - Unweighted single-source shortest path
  - BFS, DFS
- Dijkstra algorithm.
  - Weighted single-source shortest path
  - Modified BFS (with weights+priority queue)

Single-Source Short-Path Algorithm

- Breadth-first traversal (FIFO queue)
- Breadth-first traversal + priority queue = Dijkstra algorithm



Breadth-first traversal

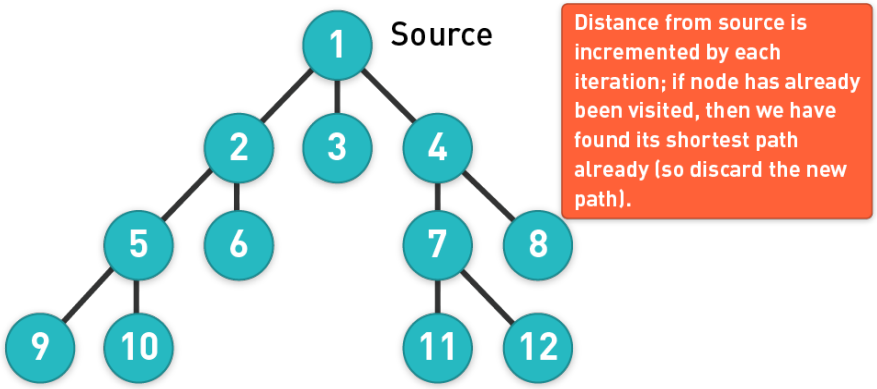
Depth-first traversal

BFS Is an O(V+E)

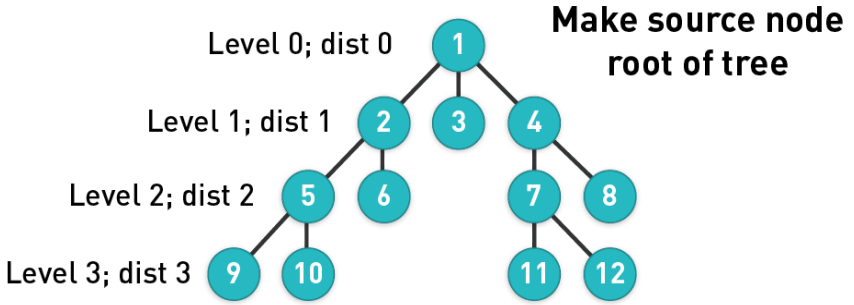
- The running time of that algorithm is O(V+E), where V is the number of vertices and E is the number of edges.
- Because it pushes each reachable vertex onto the queue and considers each outgoing edge from it once
- There can't be any faster algorithm for solving this problem, because in general the algorithm must at least look at the entire graph, which has size O(V+E).

Breadth-First Traversal

- Breadth-first traversal of a graph:
  - Is roughly analogous to level-by-level traversal of an ordered tree.
  - Start the traversal from an arbitrary vertex.
  - Visit all of its adjacent vertices.
  - Then, visit all unvisited adjacent vertices of those visited vertices in last level.
  - Continue this process, until all vertices have been visited.



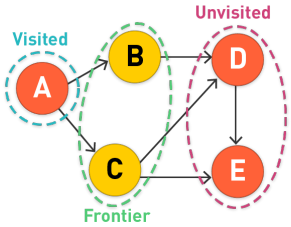
BFS (FIFO Q): Create a Treelike Structure With the Source at the Root: Visit Each Node



Breadth-first traversal

Assume Directed Graph

- Graph traversal implicitly divides the set of vertices into three sets:



1. The completed vertices: visited vertices that have already been removed from the queue
2. The frontier: visited vertices on the queue
3. The unvisited vertices: everything else

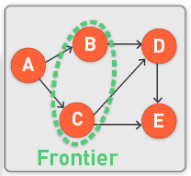
- Except for the initial vertex v0, the vertices in set 2 are always neighbors of vertices in set 1.
- Thus, the queued vertices form a frontier in the graph, separating sets 1 and 3.
- In the following BFS/DFS algorithm:
  - The expand function moves a frontier vertex (selected via FIFO/LIFO) into the completed (visited) set and then expands the frontier to include any previously unseen neighbors of the new frontier vertex.

Graph Traversal Using BFS vs. DFS: Find Shorted Path to Each Node (Unweighted Graph)

```
Let val q: queue = new_queue()
  val visited: vertexSet = create_vertexSet()
  fun expand(v: vertex) =
    let val neighbors: vertex list = Graph.outgoing(v)
      fun handle_edge(v': vertex): unit =
        if not (member(visited, v'))
        then (add(visited, v');
              push(q, v') )
        else () )
    in
      app handle_edge neighbors
    end
in
  add(visited, v0);
  expand(v0);
  while (not (empty_queue(q)) do expand(pop(q))
end
```

Start
1
2...

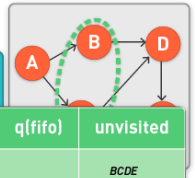
- *Move v to visited*
- *Move v' to frontier*
- *And Move v' to visited and keep track of path*



Where q is FIFO then BFS; pop() determines type of queue

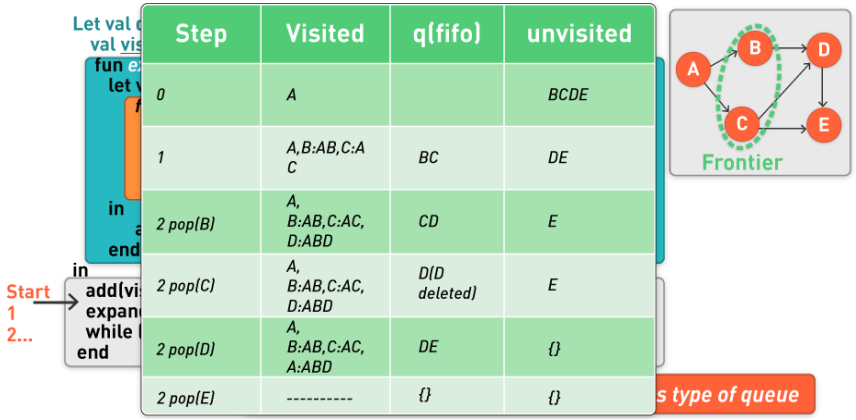
Graph Traversal Using BFS vs. DFS (cont.)

- The kind of search we get from this algorithm is determined by the pop function, which selects a vertex from a queue.
- If q is a FIFO queue, we do a breadth-first search of the graph. If q is a LIFO queue, we do a depth-first search.



	q(fifo)	unvisited
		BCDE
A	BC	DE
	CD	E
	D(D deleted)	E
	DE	{}
	{}	{}

Graph Traversal Using BFS vs. DFS: Find Shorted Path to Each Node (Unweighted Graph)



Step	Visited	q(fifo)	unvisited
0	A		BCDE
1	A,B:AB,C:A C	BC	DE
2 pop(B)	A, B:AB,C:AC, D:ABD	CD	E
2 pop(C)	A, B:AB,C:AC, D:ABD	D(D deleted)	E
2 pop(D)	A, B:AB,C:AC, A:ABD	DE	{}
2 pop(E)	----------	{}	{}

Graph Traversal Using DFS

BFS/DFS Allowed to Traverse the Graph—Track of Path Length

- The only modification needed is in expand, which adds to the frontier a newly found vertex at a distance one greater than that of its neighbor already in the frontier.

BFS and Counts Path Lengths From Source: Unweighted Graph

BFS and Counts Path Lengths From Source: Unweighted Graph

BFS Algorithm for Shortest Paths in Unweighted Graphs

```
(* unweighted single-source shortest path *)
let val q: queue = new_queue()
    val visited: vertexMap = create_vertexMap()
    (* visited maps vertex->int *)
    fun expand(v: vertex) =
     let val neighbors: vertex list = Graph.outgoing(v)
         val dist: int = valOf(get(visited, v))
         fun handle_edge(v': vertex) =
          case get(visited, v') of
            SOME(d') => () (*d' <= dist+1 *)
          | NONE => ( add(visited, v') dist+1);
                    push(q, v') )
     in
      app handle_edge neighbors
     end
in
   add(visited, v0, 0);
   expand(v0);
   while (not (empty_queue(q)) do expand(pop(q))
end
```

Single-Threaded Breadth-First Search

- One common task involving a social graph is to use it to construct a tree, starting from a particular node.
  - E.g. to construct the tree of Mary's friends and Mary's friends of friends, etc., the simplest way to do this is to perform what is called a breadth-first search (BFS).
- [You can read all about that here. this reference also includes a pseudocode implementation following a Java implementation. (The Node class is a simple bean. You can download Node.java here and Graph.java here.)]

Social Graph Example (Unweighted)

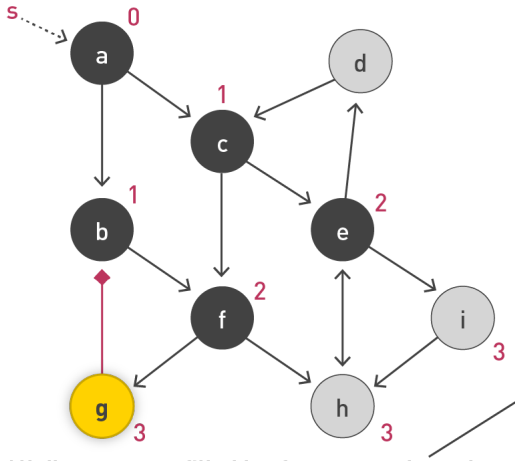
- For an introduction to graph theory, start here.
- A common use of a graph is the "social graph," e.g., your network of friends, as represented on a social network such as LinkedIn or Facebook.
    - One way to store a graph is using an adjacency list. In an adjacency list, each "node on the graph" (e.g, each person) is stored with a link to a list of the "edges emanating from that node" (e.g., their list of friends). For example :
    - Frank -> {mary, jill}
    - Jill -> {frank, bob, james}
    - Mary -> {william, joe, erin}
    - Or numerically:
        - 0-> {1, 2}
        - 2-> {3, 4, 5}
        - 1-> {6, 7, 8}

Single-Source Shortest Path: Unweighted

- In this case we are trying to find the smallest number of edges that must be traversed in order to get to every vertex in the graph from a given node.
- This is the same problem as solving the weighted version where all the weights happen to be 1.
- Unweighted graphs.
- NEXT: Talk about...
- Weighted graphs.

Breadth-First Search

Expanding node g with b. Notice the edge has a diamond at the end and not an arrow indicating we have visited it already. The node is also in black and has a shortest path associated with it of 1. So. . .



FIFO queue Q	Just-after-processing vertex
(a)	-
(a,b,c)	a
(a,b,c,f)	b
(a,b,c,f,e)	c
(a,b,c,f,e,g,h)	f
(a,b,c,f,e,g,h,d,i)	i

All distances are filled in after processing e, but we still have elements in the Q; so we continue.

That completes our example of BFS for unweighted, directed graphs.

From Unweighted to Weighted Graphs

- Now we can generalize to the problem of computing the shortest path between two vertices in a weighted graph.
- For the BFS algorithm, we can keep a visited hashmap that maps vertices to their distances from the source vertex v_0.
- Previously, when traversing an edge in our node expansion, we added 1 to the distance.
- We change expand so that instead of adding 1 to the distance, its adds the weight of the edge just traversed.
- Here is a first cut at an algorithm for weighted graphs.

BFS With Weighted Edges: Version 1

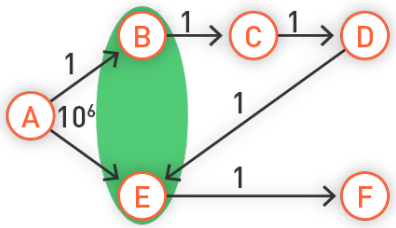
```
(* Dijkstra's Algorithm *)
let val q: queue = new_queue()
   val visited: vertexMap = create vertexMap()
   fun expand(v: vertex) =
      let  val neighbors: vertex list = Graph.outgoing(v)
           val dist: int = valOf(get(visited, v))
           fun handle_edge(v': vertex, weight: int) =
              case get(visited, v') of
           Case 1  SOME(d') =>
                     if dist+weight < d'
                     then add(visited, v', dist+weight)
                     else ()
                   NONE =>  (  add(visited, v', dist+weight);
           Case 2                push(q, v') )
      in
         app handle_edge neighbors
      end
in
S0 add(visited, v0, 0);
S1 expand(v0);
S2 while  (not  (empty_queue(q)) do expand(pop(q))
end
```

It adds the weight of the edge traversed. Here is a first cut at an algorithm.

Looking Ahead

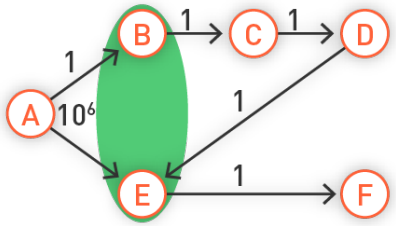
- There is a problem with this algorithm.
- To see why, I made a small challenge for you, which is up next.
- Let's see if you can invent Dijkstra's algorithm.
- Good luck!

Six-Node Graph



Step	Visited	FIFO_Q	Unvisited
0	A	A	BCDEF
1 Expand(A)—BE	A0,B1,E5	B,E	CDF
2 Expand(B)—C	A0,B1,E1000,C2	E,C	DF
3 Expand(E)—F	A0,B1,E1000,C2, F=1001	C,F	D
4 Expand(C)—D	A0,B1,E1000,C2, F=1001, D3	F,D	
5 Expand(F)—{}	A0,B1,E1000,C2, F=1001, D3	D	
6 Expand(D)—E	A0,B1,E4,C2, F=1001, D3	{}...replace E1000 with E4	
Finished	A0,B1,E4,C2, F=1001, D3	FIFO_Q is empty	

Six-Node Graph



Step	Visited	FIFO_Q	Unvisited
0	A	A	BCDEF
1 Expand(A)—BE	A0,B1,E5	B,E	CDF
2 Expand(B)—C	A0,B1,E1000,C2	E,C	DF
3 Expand(E)—F	A0,B1,E1000,C2, F=1001	C,F	D
4 Expand(C)—D	A0,B1,E1000,C2, F=1001, D3	F,D	
5 Expand(F)—{}	A0,B1,E1000,C2, F=1001, D3	D	
6 Expand(D)—E	A0,B1,E4,C2, F=1001, D3	{}...replace E1000 with E4	
Finished	A0,B1,E4,C2, F=1001, D3	FIFO_Q is empty	

BFS + 2 Fixes Gives Us Dijkstra's Shortest Path

1. New path an improvement over a possible previous one.
    - In the SOME case a check is needed to see whether the path just discovered to the vertex v' is an improvement on the previously discovered path (which had length d).
2. The queue q should not be a FIFO queue.
    - Instead, it should be a priority queue where the priorities of the vertices in the queue are their distances recorded in visited.

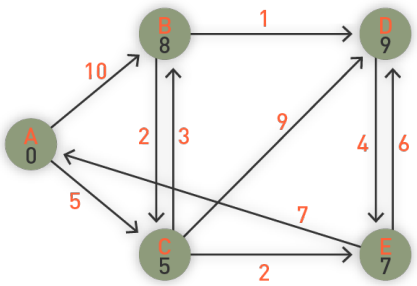
Dijkstra's Shortest Path

```
(* Dijkstra's Algorithm *)
let val q: queue = new_queue()
   val visited: vertexMap = create_vertexMap()
   fun expand(v: vertex) =
      let  val neighbors: vertex list = Graph.outgoing(v)
           val dist: int = valOf(get(visited, v))
           fun handle_edge(v': vertex, weight: int) =
              case get(visited, v') of
           Case 1  SOME(d') =>
                      if dist+weight < d'
                      then  ( add(visited, v', dist+weight);
   Update priority <-------> incr_priority(q, v', dist+weight) )
                      else  ()
           Case 2  NONE =>   (  add(visited, v', dist+weight);
                                push(q, v', dist+weight) )
      in
         app handle_edge neighbors
      end
in
   add(visited, v0, 0);
   expand(v0);
   while   (not   (empty_queue(q)) do expand(pop(q))
end
```

Two Flavors

- Single source and all destinations
  - For a given source node in the graph, the algorithm finds the shortest path between that node and every other.
- Single source and single destination
  - It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Dijkstra's Algorithm Example



End up with a data structure: Hashtable with an entry for each node and the corresponding distance from source node to each node (and path if required)

Step	Visited	PriorityQ	Unvisited
0	A		BCDE
1 Expand(A)	A	C5,B10	DE
2 pop(C)	A0,C5	E7,B8, D14	{}
3 pop(E)	A0,C5, E7	B8,D13	{}
4 pop(B)	A0,C5, E7,B8	D9	{}
5 pop(D) E13	A0,C5, E7,B8, D9	{}	{}

Step	Visited	PriorityQ
A	0	{}
B	8	AB
C	5	AC
D	9	ACD
E	7	ACE

Sundry Items on SSSP: No Negative Weights

- Short answer is that Dijkstra is a greedy algorithm, meaning we assume that we have made the best choice possible in every step of the algorithm.
- With negative edges, this is no longer possible, and thus a greedy algorithm will not suffice.
- Because of this, there are no overlapping subproblems and therefore no need for memorization of already calculated values.

Today's Lecture

- Now onto the fun part of today's lecture
- Shortest-path calculations in a weighted graph using MapReduce
- Unweighted graph
- Distributed BFS

Single-Node Computer: Priority Queue Is Key

- On a single-node computer Dijkstra's algorithm proceeds slowly down the tree one level at a time.
  - Once you are past the first level, there will be many nodes whose edges need to be examined, and in the code above this happens sequentially.
  - The key to Dijkstra's algorithm is the **global priority queue** that maintains a globally sorted list of nodes by current distance.
- Limitations of a single-node computer:
  - Single-processor machine can be slow and limited: Dijkstra’s algorithm.

Parallel MapReduce Is Stateless: Priority Queue Is Not Possible

- This is not possible in MapReduce, as the programming model does not provide a mechanism for exchanging global data (it is stateless).
  - How can we modify this to work for a huge graph and run the algorithm in parallel?
- Solution:
  - Instead, we adopt a brute-force approach known as parallel breadth-first search.

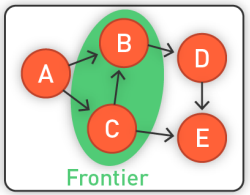
Multiple Iterations Needed

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

BFS Unweighted: Rep^t as Adjacency Lists

Graph → Adjacency Lists

Key	Value	
A	B,C	|Frontier
B	D	
C	B,E	
D	E	
E		



Each node has record: its adjacency list.

Advance the frontier at each iteration.

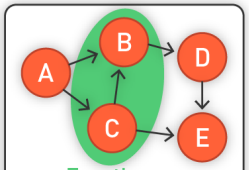
Keep track of visited.

Iterate while there is a frontier.

BFS Unweighted: From Intuition to Algorithm

Graph → Adjacency Lists

Key	Value	
A	B,C	|Frontier
B	D	
C	B,E	
D	E	
E		



Each node has record in the stream.

Init

Put source node in frontier.

Repeat

- **Map**
  - A map task processes all frontier nodes in the form.
    - Key: node *n*
    - Value: *points-to* (list of nodes reachable from *n*)
    - ∀p∈ *points-to*: emit path to node p (*p*,*np*); expand the frontier with *p*
  - All other nodes are just passed through.
- **Reduce**
  - The reduce task gathers possible paths to a given node *p* and selects one;
  - and joins it to the frontier nodes;
  - and puts the visited nodes in the visited state

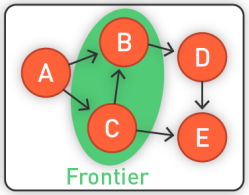
until no nodes in frontier queue.

Key	Value=Nbors	|path	|status
A	B,C	|A	|Visited
B	D	|AB	|Visited
C	B,E	|AC	|Visited
D	E	|ABD	|Visited
E		|ABDE	|Visited

BFS Unweighted: From Intuition to Algorithm

Graph → Adjacency Lists

Key	Value	
A	B,C	|Frontier
B	D	
C	B,E	
D	E	
E		



Each node has record in the stream.

Init

Put source node in frontier.

Repeat

- **Map**
  - A map task processes all frontier nodes in the form.
    - Key: node *n*
    - Value: *points-to* (list of nodes reachable from *n*)
    - ∀p∈ *points-to*: emit path to node p (*p*,*np*); expand the frontier with *p*
  - All other nodes are just passed through.
- **Reduce**
  - The reduce task gathers possible paths to a given node *p* and selects one;
  - and joins it to the frontier nodes;
  - and puts the visited nodes in the visited state

until no nodes in frontier queue.

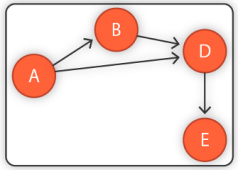
Key	Value=Nbors	path	status
A	B,C	|A	|Visited
B	D	|AB	|Visited
C	B,E	|AC	|Visited
D	E	|ABD	|Visited
E		|ABDE	|Visited

Distributed SSSP
With Unweighted Graphs

- In BFS using a FIFO queue, you would expand each node in the front sequentially.
- Here take that notion one step further and expand all nodes in the frontier in parallel.
- Reduction.
- Go from a priority queue to a parallel breadth-first algorithm.

Single-Source Shortest Path: MapReduce

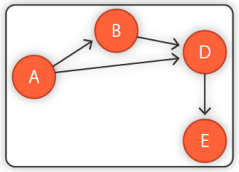
Step	Visited	q(fifo)	unvisited
0	A		BCDE
1	A,B:AB,C:AC	BC	DE
2 pop(B)	A, B:AB,C:AC, D:ABD	CD	E
2 pop(C)	A, B:AB,C:AC, D:ABD	D(D deleted)	E
2 pop(D)	A, B:AB,C:AC, A:ABD	DE	{}
2 pop(E)	----------	{}	{}



- Algorithm is based upon BFS tables that we have seen already.
  - The completed vertices: **visited** vertices that have already been removed from the queue
  - The **frontier**: visited vertices on the queue
  - The **unvisited** vertices: everything else

Single-Source Shortest Path: MapReduce

Step	Visited	q(fifo)	unvisited
0	A		BCDE
1	A,B:AB,C:AC	BC	DE
2 pop(B)	A, B:AB,C:AC, D:ABD	CD	E
2 pop(C)	A, B:AB,C:AC, D:ABD	D(D deleted)	E
2 pop(D)	A, B:AB,C:AC, A:ABD	DE	{}
2 pop(E)	----------	{}	{}

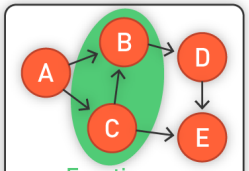


- Algorithm is based upon BFS tables that we have seen already.
  - The completed vertices: **visited** vertices that have already been removed from the queue
  - The **frontier**: visited vertices on the queue
  - The **unvisited** vertices: everything else
- And multiple calls to the same Mapper and Reducer.

BFS Unweighted: From Intuition to Algorithm

Graph → Adjacency Lists

Key	Value	
A	B,C	|Frontier
B	D	
C	B,E	
D	E	
E		



Each node has record in the stream.

Init

Put source node in frontier.

Repeat

- **Map**
  - A map task processes all frontier nodes in the form.
    - Key: node *n*
    - Value: *points-to* (list of nodes reachable from *n*)
    - ∀p∈ *points-to*: emit path to node p (*p*,*np*); expand the frontier with *p*
  - All other nodes are just passed through.
- **Reduce**
  - The reduce task gathers possible paths to a given node *p* and selects one;
  - and joins it to the frontier nodes;
  - and puts the visited nodes in the visited state

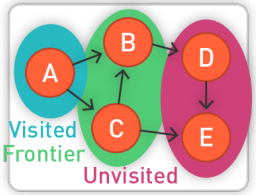
until no nodes in frontier queue.

Key	Value=Nbors	path	status
A	B,C	|A	|Visited
B	D	|AB	|Visited
C	B,E	|AC	|Visited
D	E	|ABD	|Visited
E		|ABDE	|Visited

SSSP in MapReduce: Key-Value Pair

- Single-source shortest path (SSSP)
- Represent a node in key-value format:

Key	Value		
NodeID	EDGES	DISTANCE_FROM_SOURCE	State
A	B, C	0	Q



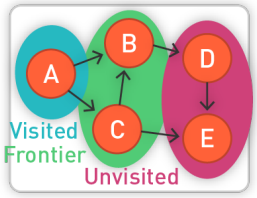
- Where ID is the node id.
- Where EDGES is a comma-delimited list of the ids of the nodes that are connected to this node.
- In the beginning, we do not know the distance and will use Integer.MAX_VALUE for marking "unknown."
- The state tells us whether or not we've seen the node before, so this starts off as unvisited. (visited|frontier|unvisited).

Key-Value Pair: Path or Distance

- Single-source shortest path (SSSP)
- Represent a node in key-value format:

Key	Value		
NodeID	EDGES	DISTANCE_FROM_SOURCE	State
A	B, C	0	Q

Key	Value		
NodeID	EDGES	PATH_FROM_SOURCE	State
A	B, C	AXXXX	Q



- Where ID is the node id.
- Where EDGES is a comma-delimited list of the ids of the nodes that are connected to this node.
- In the beginning, we do not know the distance and will use Integer.MAX_VALUE for marking "unknown."
- The state tells us whether or not we've seen the node before, so this starts off as unvisited. (visited|frontier|unvisited).

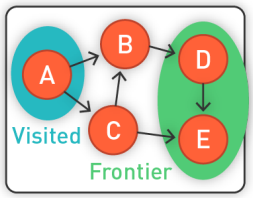
Finding the Shortest Path

- 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 ∈ S)

SSSP in MapReduce Steps

- Single-source shortest path (SSSP)
- Represent a node in key-value format:

Key	Value		
NodeID	EDGES	DISTANCE_FROM_SOURCE	State
A	B, C	0	Q

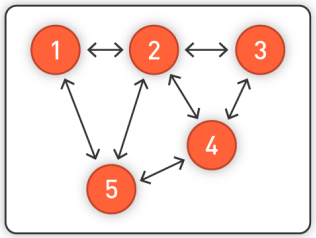


- Mapper
  - Every map iteration expands the frontier.
- Reducer
  - Every reduce iteration compresses the paths to frontier nodes to the ones that are minimum for a node in the frontier.

Example Problem: Input Adjacency Lists + Inits

- Suppose we start with the following input graph, in which we've stated that node 1 is the source (starting point) for the search.
  - Marked this one special node with distance 0 and frontier queue.
  - All other nodes are in the unvisited state U.

Key	Value		
1	2,5	0	Q
2	1,3,4,5	Integer.MAX_VALUE	U
3	2,4	Integer.MAX_VALUE	U
4	2,3,5	Integer.MAX_VALUE	U
5	1,2,4	Integer.MAX_VALUE	U

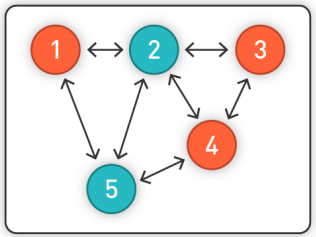


Node STATES
- *V*isited
- *Q*ueue
- *U*nvisited

SSSP First Mapper Mapper

- Mappers are responsible for expanding the frontier nodes.
  - Similar in spirit to expand(), which adds to the frontier a newly found vertex at a distance one greater than that of its neighbor already in the frontier.
  - For each frontier node, expand(); the mappers emit a new frontier node, which is with distance = distance + 1.
  - They also then emit each old frontier node with a new status; updated to visited. (Once a node has been expanded, we're done with it.)
  - Mappers also emit all unvisited nodes, with no change.
  - Note that when the mappers expand the frontier nodes and create a new node for each edge, they do not know what to write for the edges of this new node, so they leave it blank.

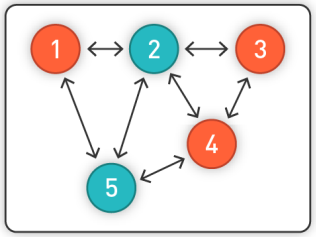
1	2,5	0	V
2	NULL	1	Q
5	NULL	1	Q
2	1,3,4,5	Integer.MAX_VALUE	U
3	2,4	Integer.MAX_VALUE	U
4	2,3,5	Integer.MAX_VALUE	U
5	1,2,4	Integer.MAX_VALUE	U



Expand the Current Frontier Using Mapper

Key	Value		
1	2,5	0	Q
2	1,3,4,5	Integer.MAX_VALUE	U
3	2,4	Integer.MAX_VALUE	U
4	2,3,5	Integer.MAX_VALUE	U
5	1,2,4	Integer.MAX_VALUE	U

Map



Frontier = {1}

Key	Value		
1	2,5	0	V
2	NULL	1	Q
5	NULL	1	Q
2	1,3,4,5	Integer.MAX_VALUE	U
3	2,4	Integer.MAX_VALUE	U
4	2,3,5	Integer.MAX_VALUE	U
5	1,2,4	Integer.MAX_VALUE	U

SSSP Reducer (Merge Candidate Paths)

If node 2 is in visited state, ignore all records in the stream and just emit the visited record for that node.

- The SSSP reducers receive all data for a given key.
  - In this case it means that they receive the data for all "copies" of each node. For example, the reducer that receives the data for key = 2 gets the following list of values.

```
2   NULL  | 1                    | Q |
2   1,3,4,5 | Integer.MAX_VALUE | U |
```

- The reducer's job is to take all this data and construct/join a new node using:
  - The nonnull list of edges
  - The minimum distance
  - The status

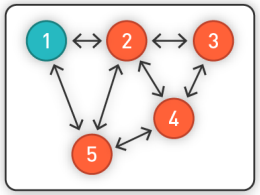
```
2   1,3,4,5  | 1 | Q |
```

SSSP MapReduce Iterations: 0th, 1st, and 2nd

Input Graph

Key	Value		
1	2,5	0	Q
2	1,3,4,5	Integer.MAX_VALUE	U
3	2,4	Integer.MAX_VALUE	U
4	2,3,5	Integer.MAX_VALUE	U
5	1,2,4	Integer.MAX_VALUE	U

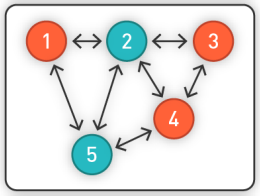
Init



Using this logic, the output from our first iteration will be

1	2,5	0	V
2	1,3,4,5	1	Q
3	2,4	Integer.MAX_VALUE	U
4	2,3,5	Integer.MAX_VALUE	U
5	1,2,4	1	Q

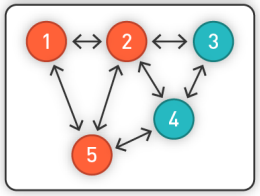
After first



The second iteration uses this as the input and outputs

1	2,5	0	V
2	1,3,4,5	1	V
3	2,4	2	Q
4	2,3,5	2	Q
5	1,2,4	1	V

After second

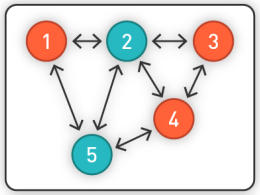


SSSP MapReduce Iterations: 1st, 2nd, and 3rd

Using this logic, the output from our first iteration will be

1	2,5	0	V
2	1,3,4,5	1	Q
3	2,4	Integer.MAX_VALUE	U
4	2,3,5	Integer.MAX_VALUE	U
5	1,2,4	1	Q

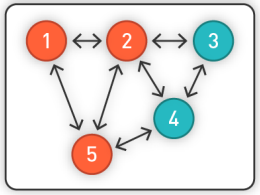
After first



The second iteration uses this as the input and outputs

1	2,5	0	V
2	1,3,4,5	1	V
3	2,4	2	Q
4	2,3,5	2	Q
5	1,2,4	1	V

After second



And the third iteration outputs:

1	2,5	0	V
2	1,3,4,5	1	V
3	2,4	2	V
4	2,3,5	2	V
5	1,2,4	1	V

After third...a fourth perhaps?

SSSP Termination: No Nodes in Q State

- Does the algorithm ever terminate?
  - Eventually, all nodes will be discovered and all edges will be considered (in a connected graph).
  - Subsequent iterations will continue to print out the same output.
- We are done when there are no output nodes that are frontier (i.e., in Q state).
- Note: If **not** all nodes in your input are actually connected to your source, you may have final output nodes that are still **unvisited**.

SSSP MapReduce Iterations: 1st, 2nd, and 3rd

Using this logic, the output from our first iteration will be 

Comparison to Dijkstra

- Dijkstra's algorithm is more efficient.
- MapReduce explores all paths in parallel.
    - Throw more hardware at the problem.

General Approach for Large Graphs

- MapReduce is adept at manipulating graphs.
  - Store graphs as adjacency lists.
- Graph algorithms for MapReduce:
  - Each map task receives a node and its outlines.
  - Map task computes 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.
- Special distributed graph libraries like Giraph, Spark Graphx are purpose-built for graph processing.