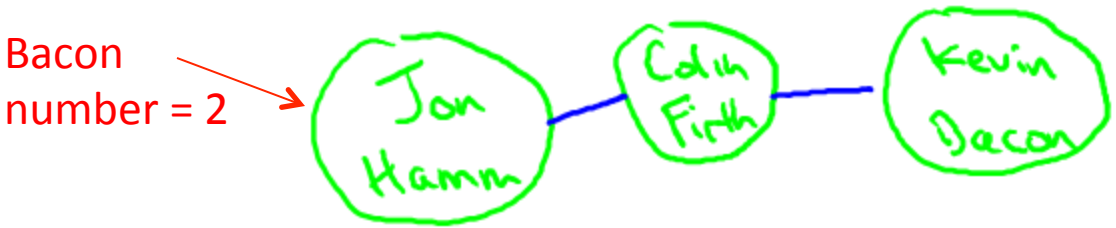
Graph Primitives

Introduction to Graph Search

Design and Analysis of Algorithms I

A Few Motivations

1. Check if a network is connected (can get to anywhere from anywhere else)



2. Driving directions
3. Formulate a plan [e.g., how to fill in a Sudoku puzzle]
   -- nodes = a partially completed puzzle -- arcs = filling in one new sequence
4. Compute the "pieces" (or "components") of a graph
   -- clustering, structure of the Web graph, etc.

Generic Graph Search

Goals : 1) find everything findable from a given start vertex

2) don't explore anything twice

Goal:
O(m+n) time

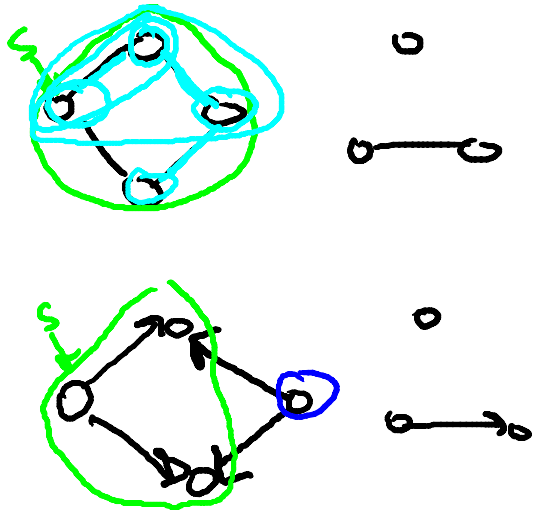
Generic Algorithm (given graph G, vertex s)

-- initially s explored, all other vertices unexplored

-- while possible : (if none, halt)

-- choose an edge (u,v) with u explored and v unexplored

-- mark v explored



Generic Graph Search (con'd)

Claim : at end of the algorithm, v explored <==> G has a path from s to v
(G undirected or directed)

Proof : (=>) easy induction on number of iterations (you check)
(<=) By contradiction. Suppose G has a path P from s to v:

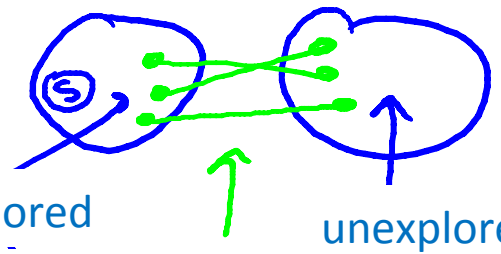
But v unexplored at end of the algorithm. Then there exists an edge (u,x) in P with u explored and x unexplored.
But then algorithm would not have terminated, contradiction.

Q.E.D.

Tim Roughgarden

BFS vs. DFS

Note : how to choose among the possibly many "frontier" edges ?



Breadth-First Search (BFS)

-- explored nodes in "layers"

-- can compute shortest paths

-- can compute connected components of an undirected graph

O(m+n) time using a queue (FIFO)

Depth-First Search (DFS)

-- explore aggressively like a maze, backtrack only when necessary

-- compute topological ordering of a directed acyclic graph

-- compute connected components in directed graphs

O(m+n) time using a stack (LIFO) (or via recursion)

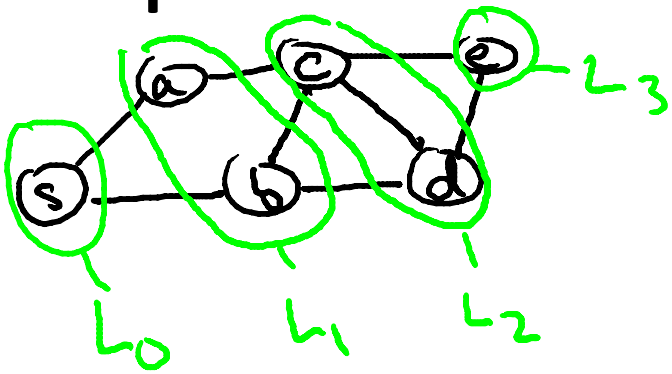
Graph Primitives

Breadth-First Search

Design and Analysis
of Algorithms I

Overview and Example

Breadth-First Search (BFS)



-- explore nodes in "layers"
-- can compute shortest paths
-- connected components of
undirected graph

Run time : O(m+n) [linear time]

The Code

BFS (graph G, start vertex s)

[all nodes initially unexplored]

-- mark s as explored

-- let Q = queue data structure (FIFO), initialized with s

-- while $Q \neq \phi$:

-- remove the first node of Q, call it v

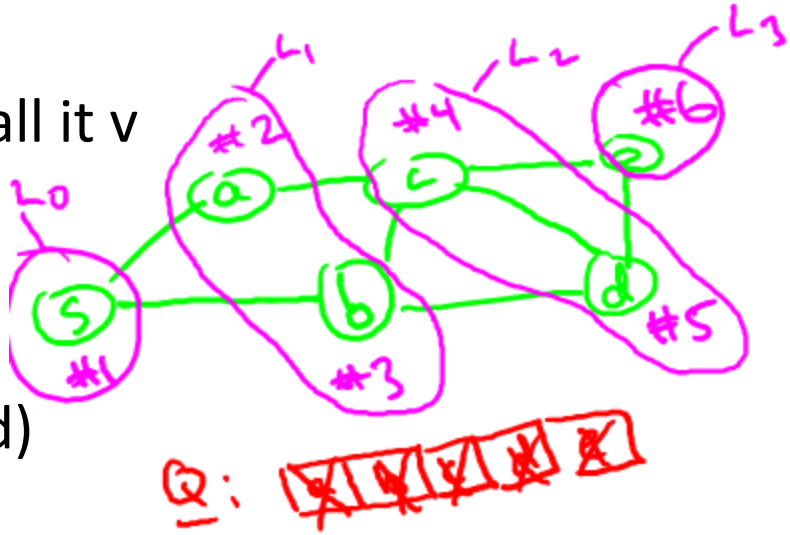
-- for each edge(v,w) :

-- if w unexplored

--mark w as explored

-- add w to Q (at the end)

O(1) time



Basic BFS Properties

Claim #1 : at the end of BFS, v explored <==>
G has a path from s to v.

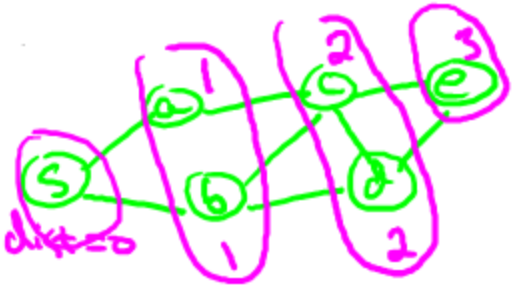
Reason : special case of the generic algorithm

Claim #2 : running time of main while loop
= $O(n_s+m_s)$, where n_s = # of nodes reachable from s
m_s = # of edges reachable from s

Reason : by inspection of code.

Application: Shortest Paths

Goal : compute dist(v), the fewest # of edges on a path from s to v.



Extra code : initialize dist(v) = $\begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s \end{cases}$

-When considering edge (v,w) :

- if w unexplored, then set dist(w) = dist(v) + 1

Claim : at termination dist(v) = i <==> v in ith layer

(i.e., shortest s-v path has i edges)

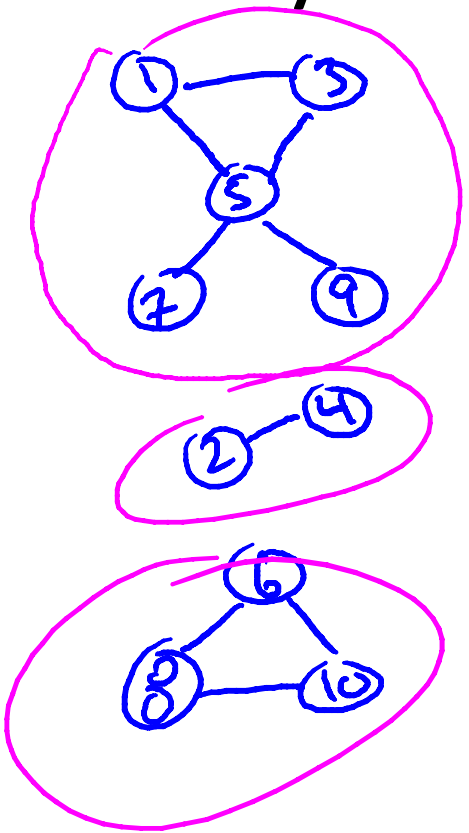
Proof Idea : every layer i node w is added to Q by a layer (i-1) node v via the edge (v,w)

Application: Undirected Connectivity

Let G = (V,E) be an undirected graph.

Connected components = the "pieces" of G.

Formal Definition : equivalence classes of the relation u<->v <==> there exists u-v path in G. [check: <-> is an equivalence relation]



Goal : compute all connected components

Why? - check if network is disconnected

- graph visualisation - clustering

Connected Components via BFS

To compute all components : (undirected case)

-- initalize all nodes as unexplored O(n)

[assume labelled 1 to n]

-- for i = 1 to n O(n)

-- if i not yet explored [in some previous BFS]

-- BFS(G,i) [discovers precisely i's connected component]

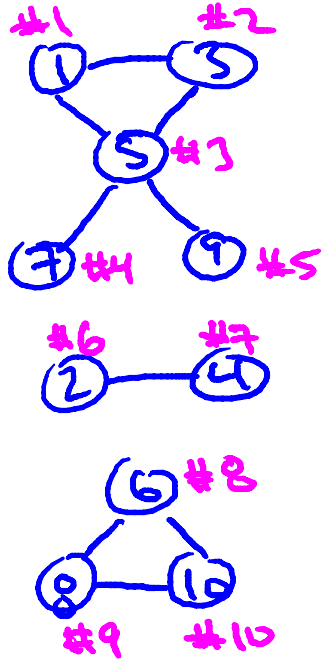
Note : finds every connected component.

Running time : O(m+n)

O(n)

O(1) per node

O(1) per edge in each BFS



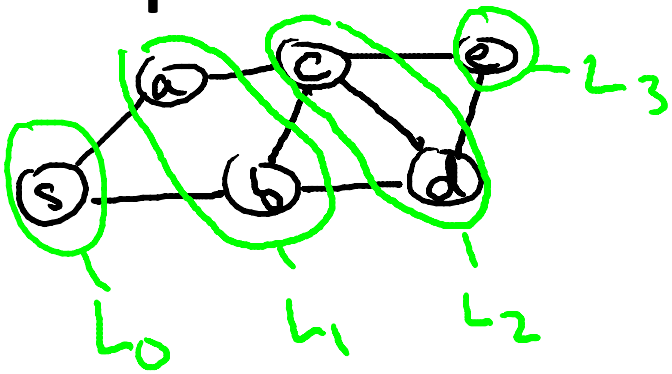
Graph Primitives

Breadth-First Search

Design and Analysis
of Algorithms I

Overview and Example

Breadth-First Search (BFS)



-- explore nodes in "layers"
-- can compute shortest paths
-- connected components of
undirected graph

Run time : O(m+n) [linear time]

The Code

BFS (graph G, start vertex s)

[all nodes initially unexplored]

-- mark s as explored

-- let Q = queue data structure (FIFO), initialized with s

-- while $Q \neq \phi$:

-- remove the first node of Q, call it v

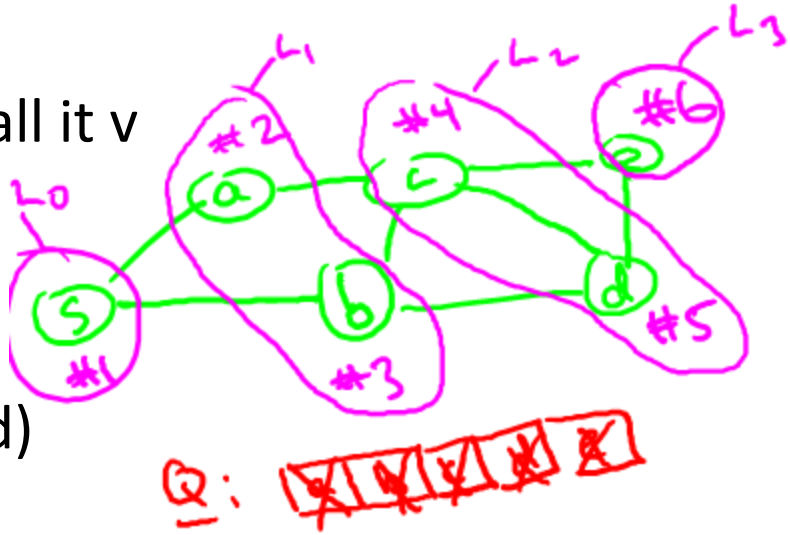
-- for each edge(v,w) :

-- if w unexplored

--mark w as explored

-- add w to Q (at the end)

O(1) time



Basic BFS Properties

Claim #1 : at the end of BFS, v explored <==>
G has a path from s to v.

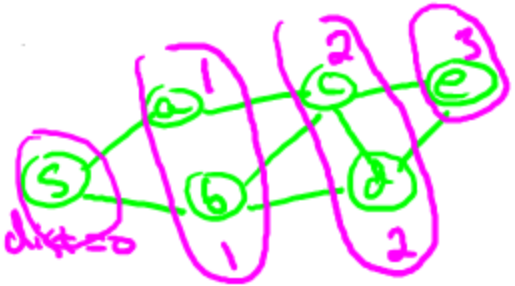
Reason : special case of the generic algorithm

Claim #2 : running time of main while loop
= $O(n_s+m_s)$, where n_s = # of nodes reachable from s
m_s = # of edges reachable from s

Reason : by inspection of code.

Application: Shortest Paths

Goal : compute dist(v), the fewest # of edges on a path from s to v.



Extra code : initialize dist(v) = $\begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s \end{cases}$

-When considering edge (v,w) :

- if w unexplored, then set dist(w) = dist(v) + 1

Claim : at termination dist(v) = i <==> v in ith layer

(i.e., shortest s-v path has i edges)

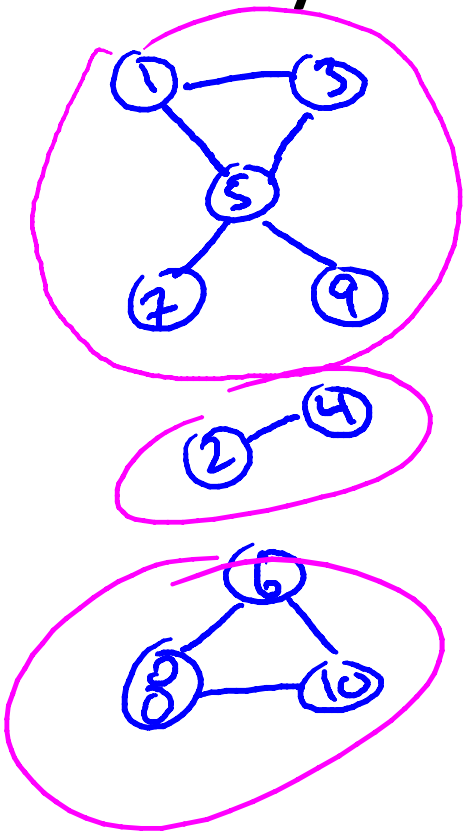
Proof Idea : every layer i node w is added to Q by a layer (i-1) node v via the edge (v,w)

Application: Undirected Connectivity

Let G = (V,E) be an undirected graph.

Connected components = the "pieces" of G.

Formal Definition : equivalence classes of the relation u<->v <==> there exists u-v path in G. [check: <-> is an equivalence relation]



Goal : compute all connected components

Why? - check if network is disconnected

- graph visualisation - clustering

Connected Components via BFS

To compute all components : (undirected case)

-- initalize all nodes as unexplored O(n)

[assume labelled 1 to n]

-- for i = 1 to n O(n)

-- if i not yet explored [in some previous BFS]

-- BFS(G,i) [discovers precisely i's connected component]

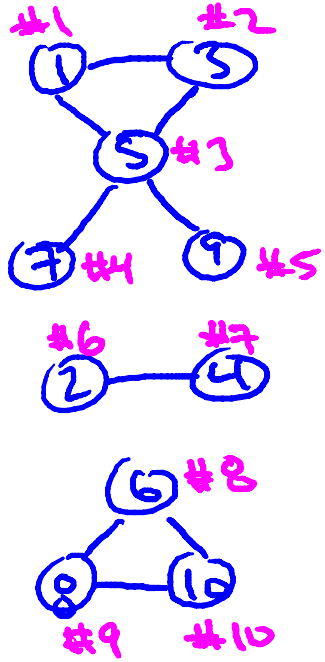
Note : finds every connected component.

Running time : O(m+n)

O(n)

O(1) per node

O(1) per edge in each BFS



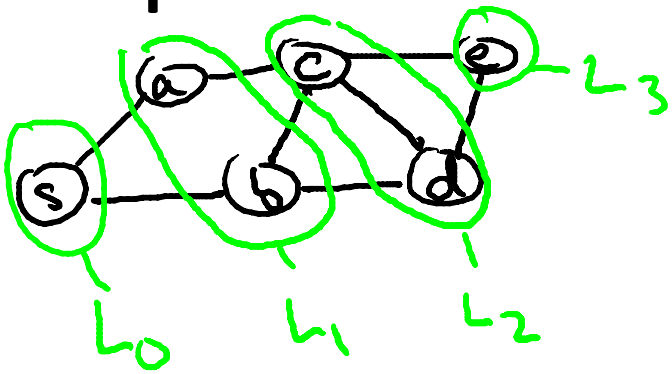
Graph Primitives

Breadth-First Search

Design and Analysis
of Algorithms I

Overview and Example

Breadth-First Search (BFS)



-- explore nodes in "layers"
-- can compute shortest paths
-- connected components of
undirected graph

Run time : O(m+n) [linear time]

The Code

BFS (graph G, start vertex s)

[all nodes initially unexplored]

-- mark s as explored

-- let Q = queue data structure (FIFO), initialized with s

-- while $Q \neq \phi$:

-- remove the first node of Q, call it v

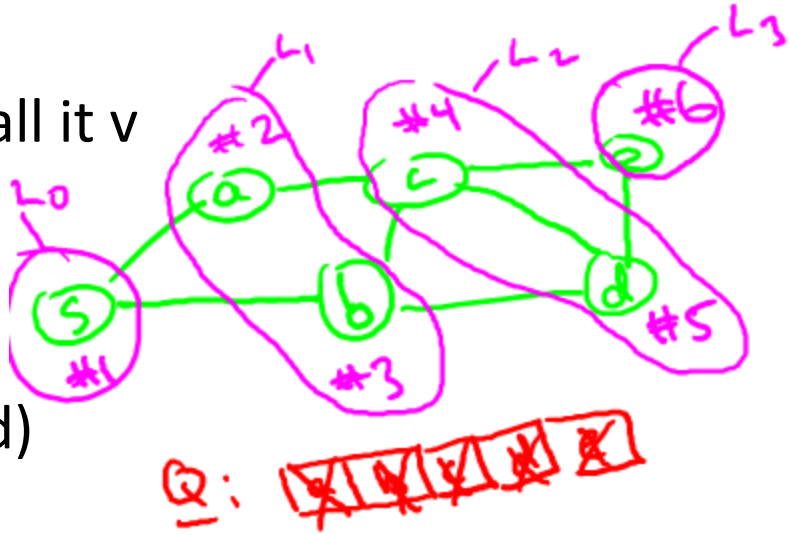
-- for each edge(v,w) :

-- if w unexplored

--mark w as explored

-- add w to Q (at the end)

O(1) time



Basic BFS Properties

Claim #1 : at the end of BFS, v explored <==>
G has a path from s to v.

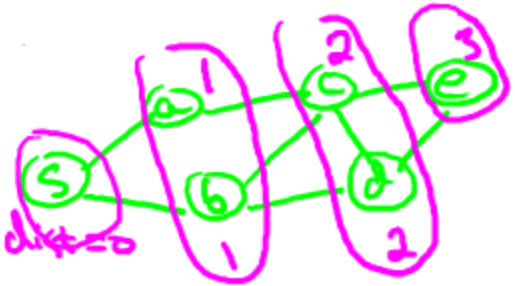
Reason : special case of the generic algorithm

Claim #2 : running time of main while loop
= $O(n_s+m_s)$, where n_s = # of nodes reachable from s
m_s = # of edges reachable from s

Reason : by inspection of code.

Application: Shortest Paths

Goal : compute dist(v), the fewest # of edges on a path from s to v.



Extra code : initialize dist(v) = $\begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s \end{cases}$

-When considering edge (v,w) :

- if w unexplored, then set dist(w) = dist(v) + 1

Claim : at termination dist(v) = i <==> v in ith layer

(i.e., shortest s-v path has i edges)

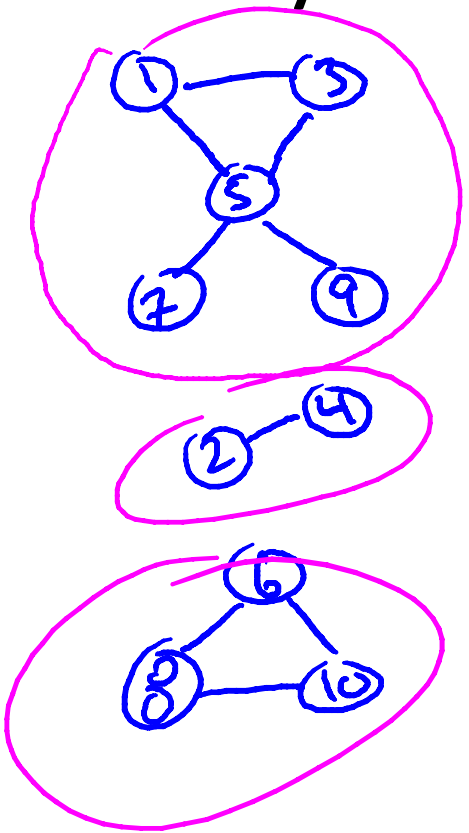
Proof Idea : every layer i node w is added to Q by a layer (i-1) node v via the edge (v,w)

Application: Undirected Connectivity

Let G = (V,E) be an undirected graph.

Connected components = the "pieces" of G.

Formal Definition : equivalence classes of the relation u<->v <==> there exists u-v path in G. [check: <-> is an equivalence relation]



Goal : compute all connected components

Why? - check if network is disconnected

- graph visualisation - clustering

Connected Components via BFS

To compute all components : (undirected case)

-- initalize all nodes as unexplored O(n)

[assume labelled 1 to n]

-- for i = 1 to n O(n)

-- if i not yet explored [in some previous BFS]

-- BFS(G,i) [discovers precisely i's connected component]

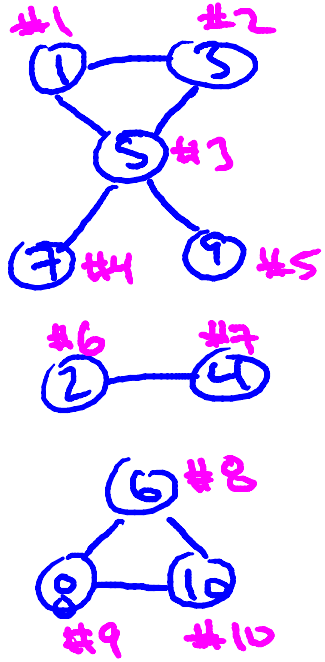
Note : finds every connected component.

Running time : O(m+n)

O(n)

O(1) per node

O(1) per edge in each BFS



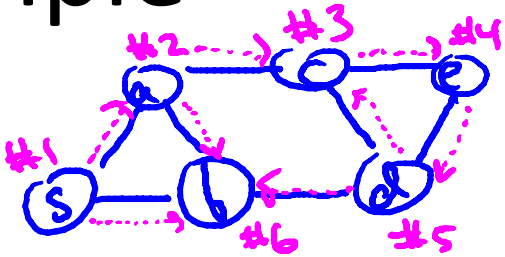
Graph Primitives

Depth-First Search

Design and Analysis
of Algorithms I

Overview and Example

Depth-First Search (DFS) : explore aggressively, only backtrack when necessary.



-- also computes a topological ordering of a directed acyclic graph
-- and strongly connected components of directed graphs

Run Time : O(m+n)

The Code

Exercise : mimic BFS code, use a stack instead of a queue [+ some other minor modifications]

Recursive version : DFS(graph G, start vertex s)

-- mark s as explored

-- for every edge (s,v) :

-- if v unexplored

-- DFS(G,v)

Basic DFS Properties

Claim #1 : at the end of the algorithm, v marked as explored
<==> there exists a path from s to v in G.

Reason : particular instantiation of generic search procedure

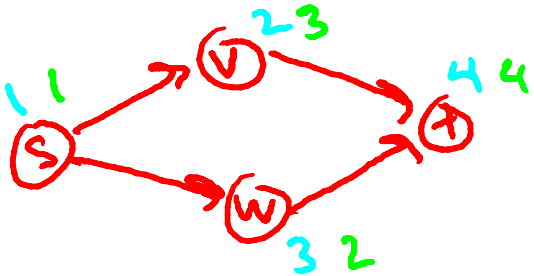
Claim #2 : running time is $O(n_s+m_s)$,
where n_s = # of nodes reachable from s
m_s = # of edges reachable from s

Reason : looks at each node in the connected component of s at most once, each edge at most twice.

Application: Topological Sort

Definition : A topological ordering of a directed graph G is a labeling f of G's nodes such that:

1. The f(v)'s are the set {1,2,..,n}
2. $(u,v) \in G => f(u) < f(v)$



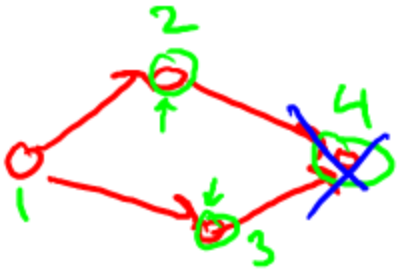
Motivation : sequence tasks while respecting all precedence constraints.

Note : G has directed cycle => no topological ordering

Theorem : no directed cycle => can compute topological ordering in O(m+n) time.

Straightforward Solution

Note : every directed acyclic graph has a sink vertex.



Reason : if not, can keep following outgoing arcs to produce a directed cycle.

To compute topological ordering :

-- let v be a sink vertex of G
-- set f(v) = n
-- recurse on G-{v}

Why does it work? : when v is assigned to position i, all outgoing arcs already deleted => all lead to later vertices in ordering.

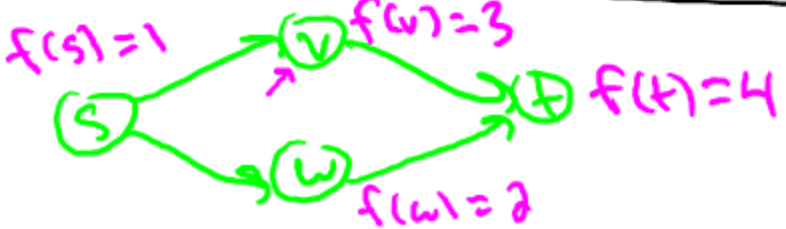
Topological Sort via DFS (Slick)

DFS-Loop (graph G)
-- mark all nodes unexplored
-- current-label = n [to keep track of ordering]

-- for each vertex
  -- if v not yet explored [in previous DFS call]

    -- DFS(G,v)

DFS(graph G, start vertex s)
-- for every edge (s,v)
  -- if v not yet explored
    -- mark v explored
    -- DFS(G,v)
-- set f(s) = current_label
-- current_label = current_label-1



Topological Sort via DFS (con'd)

Running Time : O(m+n).

Reason : O(1) time per node, O(1) time per edge.

Correctness : need to show that if (u,v) is an edge, then f(u) < f(v)

(since no directed cycles)

Case 1 : u visited by DFS before v => recursive call corresponding to v finishes before that of u (since DFS).
$\Rightarrow f(v) > f(u)$

Case 2 : v visited before u => v's recursive call finishes before u's even starts. => f(v) > f(u)

Q.E.D.

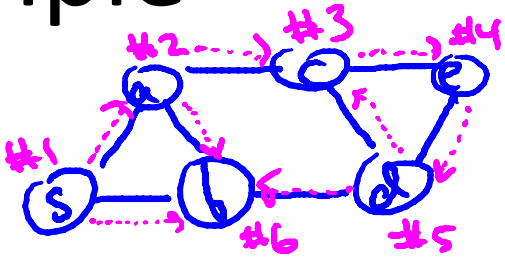
Graph Primitives

Depth-First Search

Design and Analysis
of Algorithms I

Overview and Example

Depth-First Search (DFS) : explore aggressively, only backtrack when necessary.



-- also computes a topological ordering of a directed acyclic graph
-- and strongly connected components of directed graphs

Run Time : O(m+n)

The Code

Exercise : mimic BFS code, use a stack instead of a queue [+ some other minor modifications]

Recursive version : DFS(graph G, start vertex s)

-- mark s as explored

-- for every edge (s,v) :

-- if v unexplored

-- DFS(G,v)

Basic DFS Properties

Claim #1 : at the end of the algorithm, v marked as explored
<==> there exists a path from s to v in G.

Reason : particular instantiation of generic search procedure

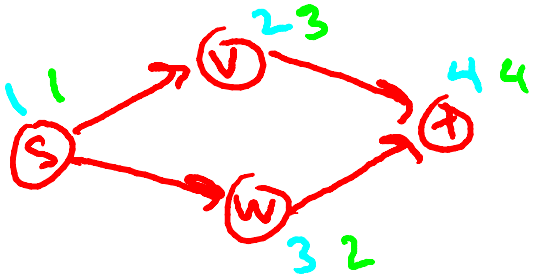
Claim #2 : running time is $O(n_s+m_s)$,
where n_s = # of nodes reachable from s
m_s = # of edges reachable from s

Reason : looks at each node in the connected component of s at most once, each edge at most twice.

Application: Topological Sort

Definition : A topological ordering of a directed graph G is a labeling f of G's nodes such that:

1. The f(v)'s are the set {1,2,..,n}
2. $(u,v) \in G => f(u) < f(v)$



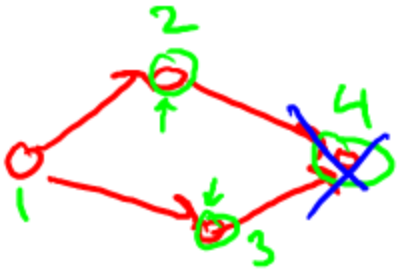
Motivation : sequence tasks while respecting all precedence constraints.

Note : G has directed cycle => no topological ordering

Theorem : no directed cycle => can compute topological ordering in O(m+n) time.

Straightforward Solution

Note : every directed acyclic graph has a sink vertex.



Reason : if not, can keep following outgoing arcs to produce a directed cycle.

To compute topological ordering :

-- let v be a sink vertex of G
-- set f(v) = n
-- recurse on G-{v}

Why does it work? : when v is assigned to position i, all outgoing arcs already deleted => all lead to later vertices in ordering.

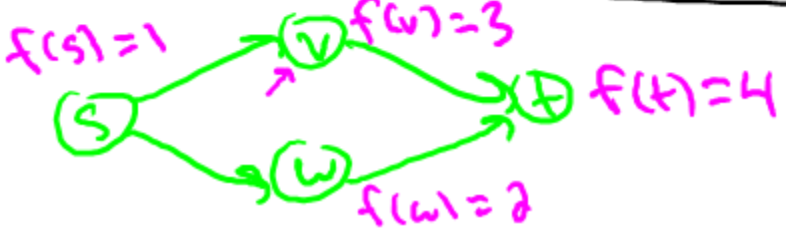
Topological Sort via DFS (Slick)

DFS-Loop (graph G)
-- mark all nodes unexplored
-- current-label = n [to keep track of ordering]

-- for each vertex
  -- if v not yet explored [in previous DFS call]

    -- DFS(G,v)

DFS(graph G, start vertex s)
-- for every edge (s,v)
  -- if v not yet explored
    -- mark v explored
    -- DFS(G,v)
-- set f(s) = current_label
-- current_label = current_label-1



Topological Sort via DFS (con'd)

Running Time : O(m+n).

Reason : O(1) time per node, O(1) time per edge.

Correctness : need to show that if (u,v) is an edge, then f(u) < f(v)

(since no directed cycles)

Case 1 : u visited by DFS before v => recursive call corresponding to v finishes before that of u (since DFS).
$\Rightarrow f(v) > f(u)$

Case 2 : v visited before u => v's recursive call finishes before u's even starts. => f(v) > f(u)

Q.E.D.

Design and Analysis of Algorithms I

Graph Primitives

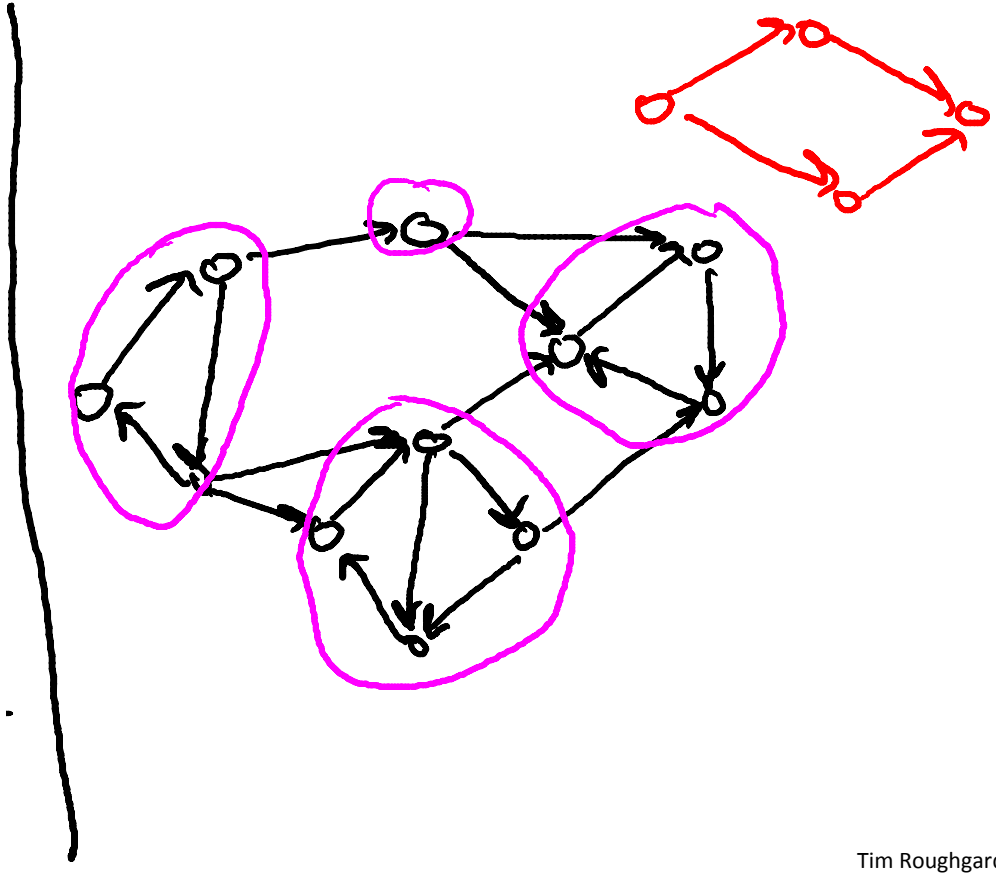
An O(m+n) Algorithm for Computing Strong Components

Strongly Connected Components

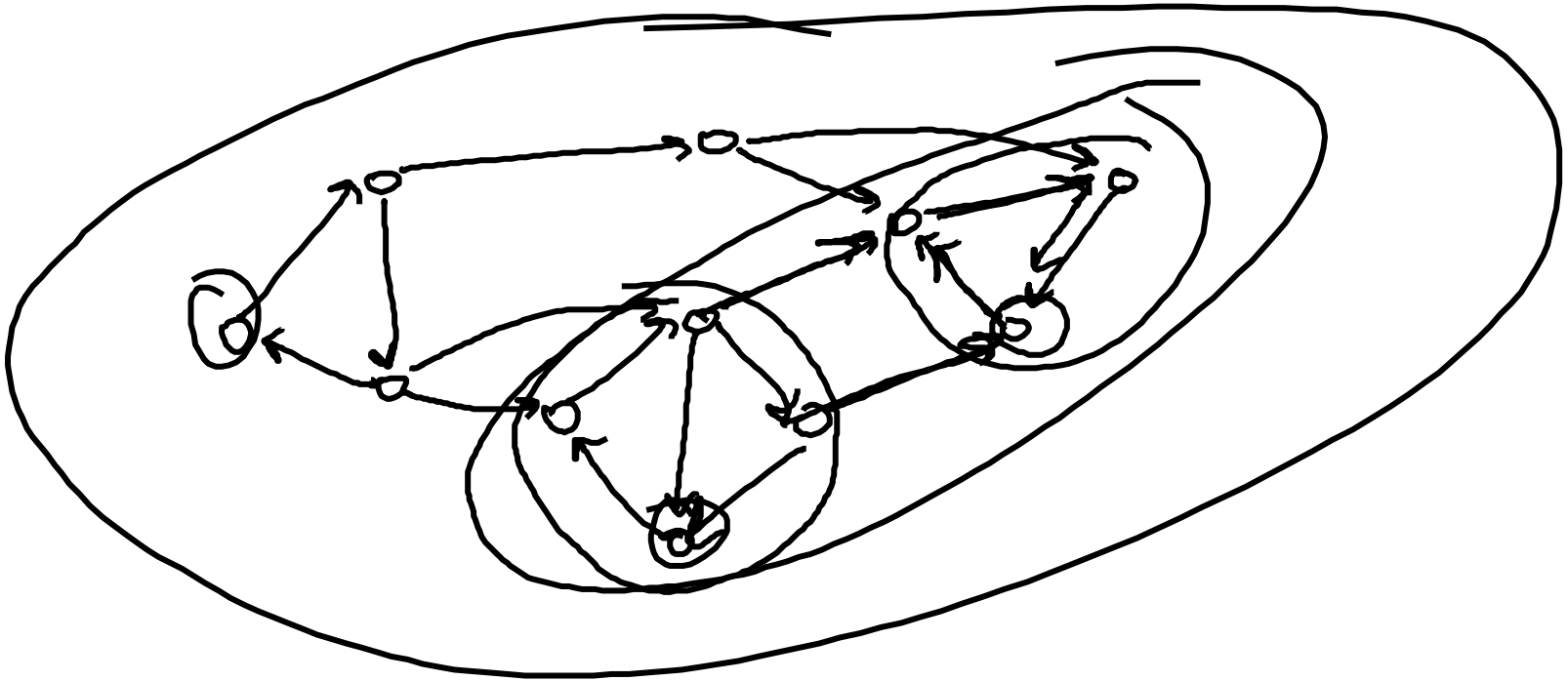
Formal Definition : the strongly connected components (SCCs) of a directed graph G are the equivalence classes of the relation

u<->v <==> there exists a path u->v and a path v->u in G

You check : <-> is an equivalence relation



Why Depth-First Search?



Tim Roughgarden

Kosaraju's Two-Pass Algorithm

Theorem : can compute SCCs in O(m+n) time.

Algorithm : (given directed graph G)

1. Let Grev = G with all arcs reversed
2. Run DFS-Loop on Grev

   Let f(v) = "finishing time" of each v in V

1. Run DFS-Loop on G

   processing nodes in decreasing order of finishing times

[SCCs = nodes with the same "leader"]

Goal : compute "magical ordering" of nodes

Goal : discover the SCCs one-by-one

DFS-Loop

DFS-Loop (graph G)

Global variable t = 0

[# of nodes processed so far]

Global variable s = NULL

[current source vertex]

Assume nodes labeled 1 to n

For i = n down to 1

    if i not yet explored

        s := i

        DFS(G,i)

For finishing times in 1st pass

For leaders in 2nd pass

DFS (graph G, node i)

-- mark i as explored

-- set leader(i) := node s

-- for each arc (i,j) in G :

        -- if j not yet explored

            -- DFS(G,j)

-- t++

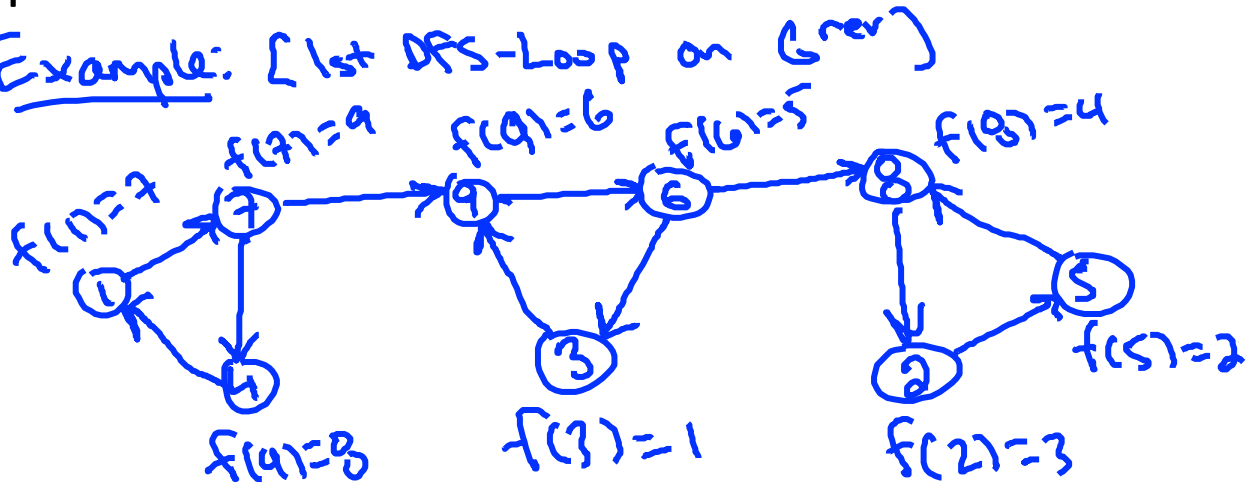
-- set f(i) := t

i's finishing time

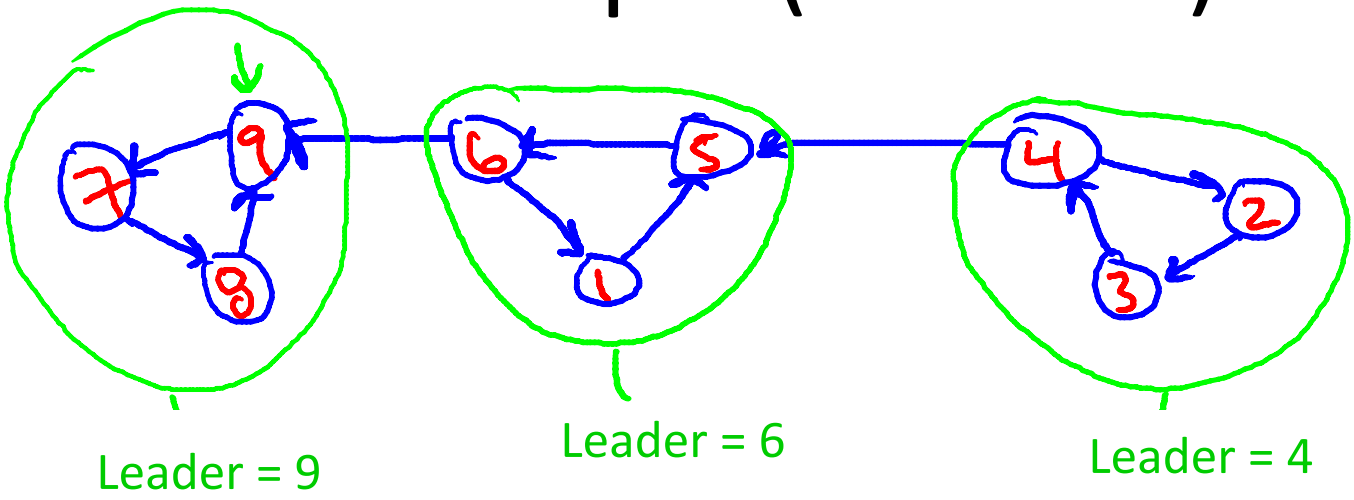
For rest of DFS-Loop

Only one of the following is a possible set of finishing times for the nodes 1,2,3,…,9, respectively, when the DFS-Loop subroutine is executed on the graph below. Which is it?

- 9,8,7,6,5,4,3,2,1
- 1,7,4,9,6,3,8,2,5
- 1,7,9,6,8,2,5,3,4
- 7,3,1,8,2,5,9,4,6



Example (2nd Pass)



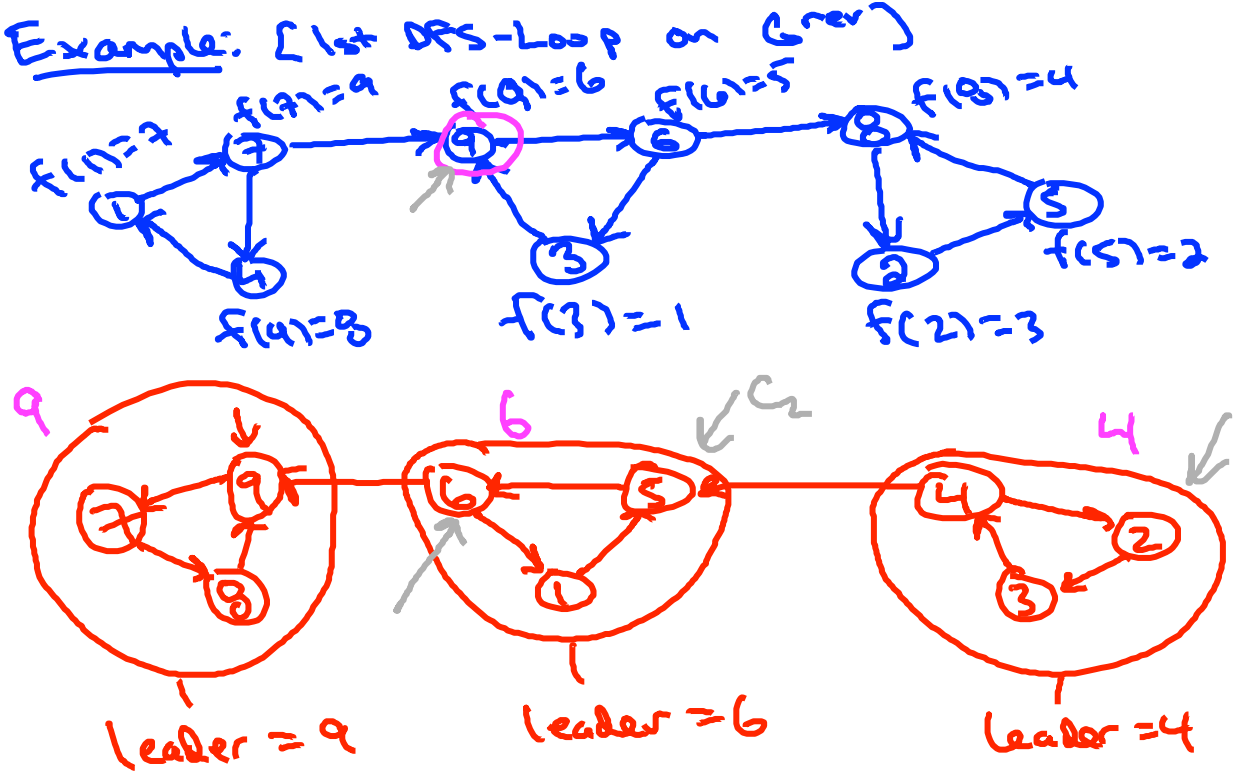
Running Time : 2*DFS = O(m+n)

Graph Primitives

Correctness of Kosaraju's Algorithm

Design and Analysis of Algorithms I

Example Recap



Observation

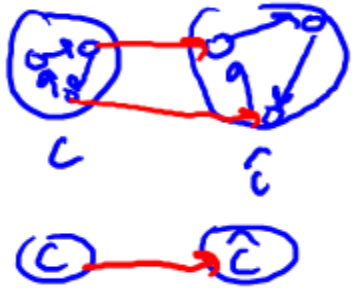
Claim : the SCCs of a directed graph G induce an acyclic "meta-graph":

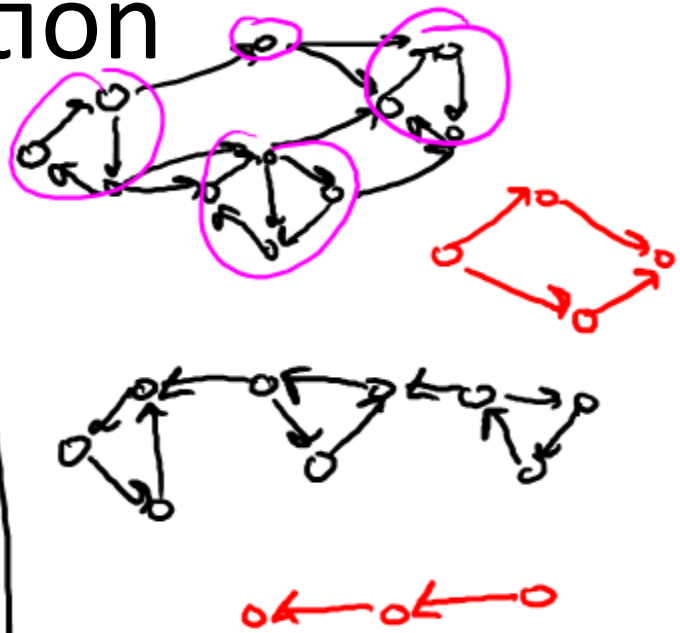
-- meta-nodes = the SCCs $C_1,..,C_k$ of G

-- $\exists\ arc\ C \to \hat{C} <=> \exists\ arc\ (i,j) \in G$

$with\ i \in C,\ j \in \hat{C}$

Why acyclic ? : a cycle of SCCs would collapse into one.





What how are the SCC of the original graph G and its reversal G^{rev} related?

- In general, they are unrelated.
- Every SCC of G is contained in an SCC of G^{rev}, but the converse need not hold.
- Every *SCC of G^{rev} is contained in an SCC of G, but the converse need not hold.*
- They are exactly the same.

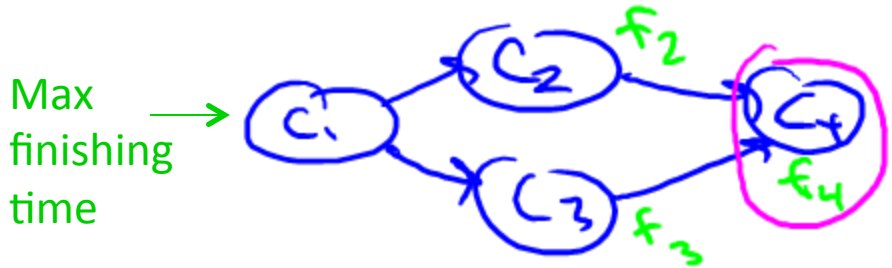
Key Lemma

Lemma : consider two "adjacent" SCCs in G:

Let f(v) = finishing times of DFS-Loop in Grev

Then : $max_{v \in C_1} f(v) < max_{v \in C_2} f(v)$

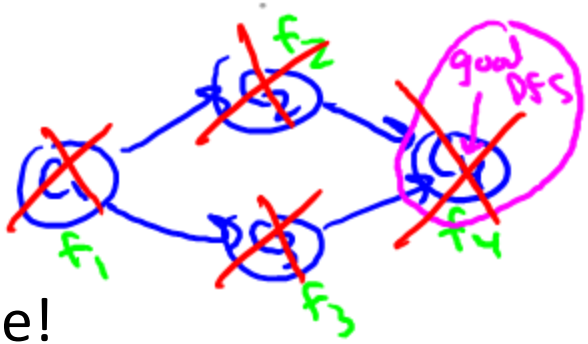
Corollary : maximum f-value of G must lie in a "sink SCC"



$f_1 < f_2, f_3 < f_4$

Correctness Intuition

(see notes for formal proof)



By Corollary : 2nd pass of DFS-Loop begins somewhere in a sink SCC C*.

⇒First call to DFS discovers C* and nothing else!

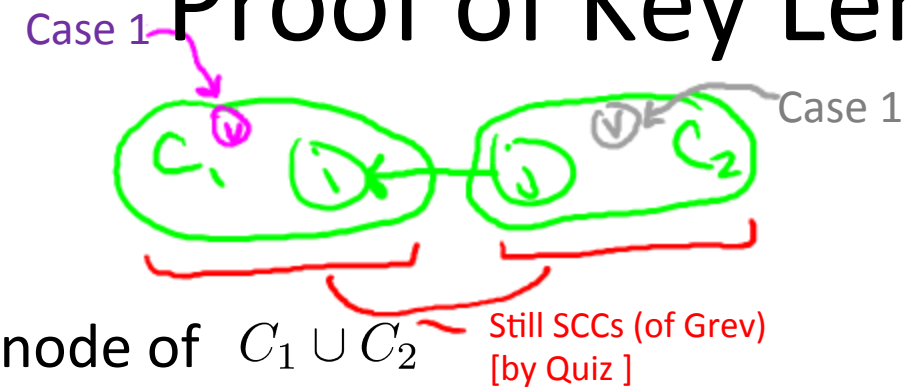
⇒Rest of DFS-Loop like recursing on G with C* deleted

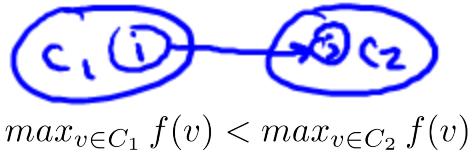
[starts in a sink node of G-C*]

⇒ successive calls to DFS(G,i) "peel off" the SCCs one by one

[in reverse topological order of the "meta-graph" of SCCs]

Proof of Key Lemma





In Grev :

Let v = 1st node of $C_1 \cup C_2$ reached by 1st pass of DFS-Loop (on Grev)

Case 1 [$v \in C_1$] : all of C_1 explored before C_2 ever reached.

Reason : no paths from C_1 to C_2 (since meta-graph is acyclic)

$\Rightarrow$All f-values in C_1 less than all f-values in C_2

Case 2 [$v \in C_2$] : DFS(Grev, v) won't finish until all of $C_1 \cup C_2$ completely explored => f(v) > f(w) for all w in C_1

Q.E.D.

Tim Roughgarden

Graph Primitives

Structure of the Web

Design and Analysis
of Algorithms I

The Web graph

-Vertices = Web pages
- (directed) edges = hyperlinks

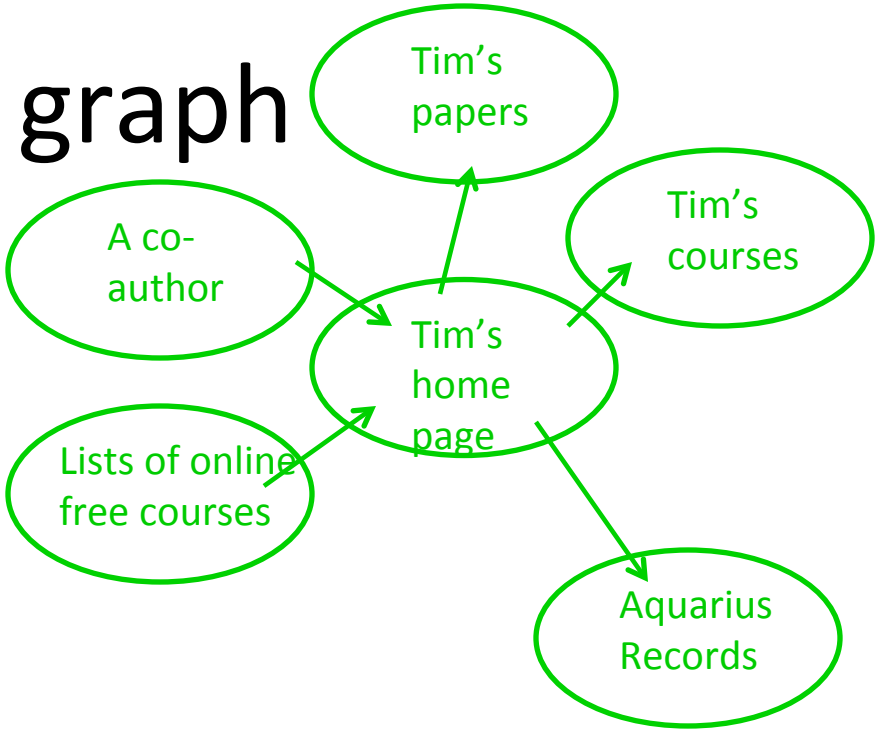
Question : what does the web graph

look like ?

(assume you've already "crawled" it)

Size : ~ 200 million nodes, ~ 1 billion edges

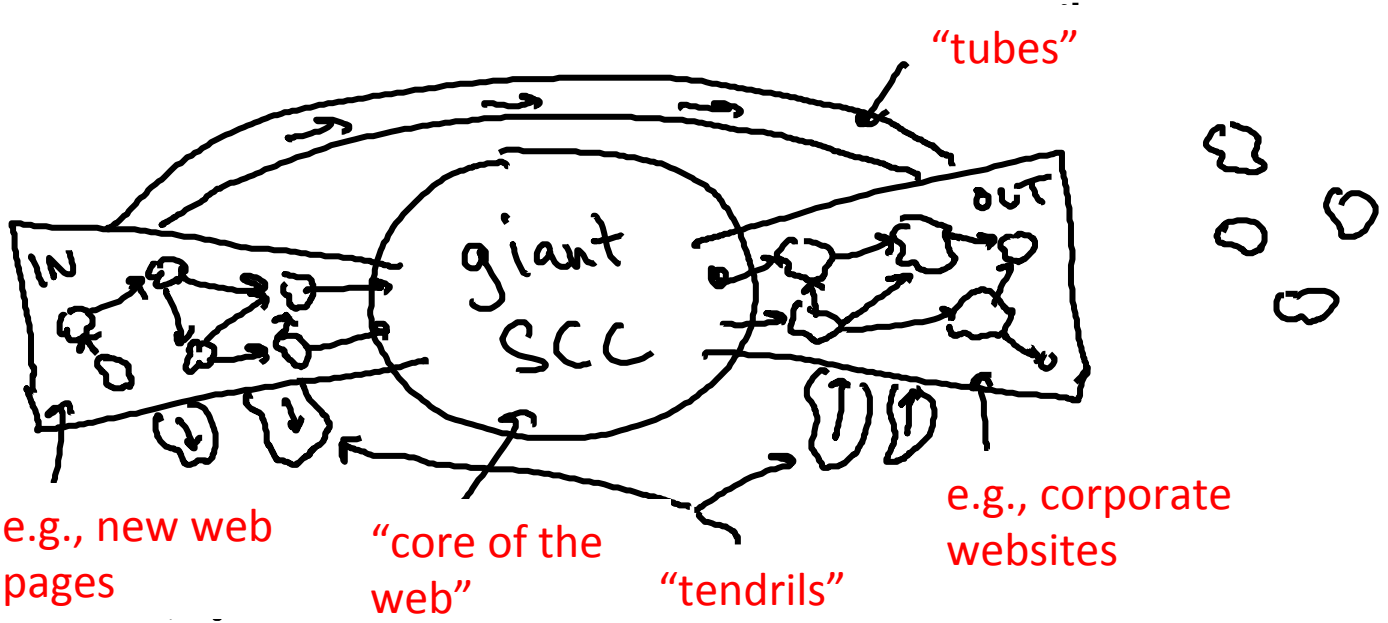
Reference : [Broder et al WWW 2000]
computed the SCCs of the Web graph.



ETC.
ETC.

(pre map-reduce/hadoop)

The Bow Tie


"tubes"
IN
giant
SCC
OUT
e.g., new web pages
"core of the web"
"tendrils"
e.g., corporate websites

Main Findings

1. All 4 parts (giant, IN, OUT, tubes + tendrils) have roughly the same size

2. Within CORE, very well connected (has the “small world” property) [Milgram]

3. Outside, surprisingly poorly connected

Modern Web Research

1. Temporal aspects – how is the web graph evolving over time ?
2. Informational aspects – how does new information propagate throughout the Web (or blogosphere, or Twitter, etc.)
3. Finer-grained structure – how to define and compute "communities" in information and social networks ?

   Recommended Reading : Easley + Kleinberg, "Networks, Crowds, & Markets"