CSE 417: Algorithms

Graphs and Graph Algorithms
Larry Ruzzo
Goals

Graphs: defns, examples, utility, terminology
Representation: input, internal
Traversal: Breadth- & Depth-first search
Five Graph Algorithms:
  Connected components
  Shortest Paths
  Topological sort
  Bipartiteness
  Articulation points
Graphs

An extremely important formalism for representing (binary) relationships

Objects: "vertices," aka "nodes"

Relationships between pairs:
  "edges," aka "arcs"

Formally, a graph $G = (V, E)$ is a pair of sets, $V$ the vertices and $E$ the edges
Objects & Relationships

The Kevin Bacon Game:
Obj: Actors
Rel: Two are related if they've been in a movie together

Exam Scheduling:
Obj: Classes
Rel: Two are related if they have students in common

Traveling Salesperson Problem:
Obj: Cities
Rel: Two are related if can travel directly between them
Undirected Graph \( G = (V, E) \)
Undirected Graph $G = (V, E)$
Undirected Graph $G = (V, E)$
Undirected Graph \( G = (V, E) \)
Undirected Graph \( G = (V,E) \)
Graphs don't live in Flatland

Geometrical drawing is mentally convenient, but mathematically irrelevant: 4 drawings, 1 graph.
Directed Graph $G = (V,E)$
Directed Graph $G = (V,E)$
Directed Graph $G = (V,E)$
Directed Graph $G = (V,E)$
Directed Graph $G = (V,E)$
Specifying undirected graphs as input

What are the vertices?
Maybe explicitly list them:
{"A", "7", "3", "4"}

What are the edges?
Either, set of edges
{{A,3}, {7,4}, {4,3}, {4,A}}
Or, (symmetric) adjacency matrix:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>7</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Specifying directed graphs as input

What are the vertices?
   Maybe explicitly list them:
   {"A", "7", "3", "4"}  

What are the edges?
   Either, set of directed edges:
   {(A,4), (4,7), (4,3), (4,A), (A,3)}
   Or, (nonsymmetric) adjacency matrix:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>7</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
# Vertices vs # Edges

Let G be an undirected graph with \( n \) vertices and \( m \) edges. How are \( n \) and \( m \) related?

Since every edge connects two different vertices (no loops), and no two edges connect the same two vertices (no multi-edges), it must be true that:

\[
0 \leq m \leq n(n-1)/2 = O(n^2)
\]

More Cool Graph Lingo

A graph is called **sparse** if \( m \ll n^2 \), otherwise it is **dense**

Boundary is somewhat fuzzy; \( O(n) \) edges is certainly sparse, \( \Omega(n^2) \) edges is dense.

Sparse graphs are common in practice

E.g., all planar graphs are sparse \((m \leq 3n-6, \text{ for } n \geq 3)\)

Q: which is a better run time, \( O(n+m) \) or \( O(n^2) \)?

A: \( O(n+m) = O(n^2) \), but \( n+m \) usually way better!
Representing Graph $G = (V,E)$

Vertex set $V = \{v_1, \ldots, v_n\}$

Adjacency Matrix $A$

$A[i,j] = 1$ iff $(v_i, v_j) \in E$

Space is $n^2$ bits

Advantages:

$O(1)$ test for presence or absence of edges.

Disadvantages: inefficient for sparse graphs, both in storage and access
Representing Graph \( G=(V,E) \)

\( n \) vertices, \( m \) edges

**Adjacency List:**

- \( O(n+m) \) words

**Advantages:**

- Compact for sparse graphs
- Easily see all edges

**Disadvantages**

- More complex data structure
- No \( O(1) \) edge test
Representing Graph \( G=(V,E) \)

- \( n \) vertices, \( m \) edges

Adjacency List:
- \( O(n+m) \) words

Back- and cross pointers allow easier traversal and deletion of edges, *if needed*, but don't bother if not:
- more work to build,
- more storage overhead (\(~6m\) pointers)
Graph Traversal

Learn the basic structure of a graph
"Walk," via edges, from a fixed starting vertex $s$ to all vertices reachable from $s$

Being *orderly* helps. Two common ways:
  
  Breadth-First Search
  
  Depth-First Search
Breadth-First Search

Completely explore the vertices in order of their distance from $s$

Naturally implemented using a queue
BFS(s) Implementation

Global initialization: mark all vertices "undiscovered"

BFS(s)
mark s "discovered"
queue = { s }
while queue not empty
    u = remove_first(queue)
    for each edge {u,x}
        if (x is undiscovered)
            mark x discovered
            append x on queue
    mark u fully explored

Exercise: modify code to number vertices & compute level numbers
BFS(v)
BFS(v)

Queue:
2 3
BFS(v)

Queue: 3 4
BFS(v)

Queue: 4 5 6 7
BFS(ν)

Queue: 5 6 7 8 9
BFS(v)

Queue: 8 9 10 11
BFS(v)

Queue: 10 11 12 13
BFS(v)

Queue:

1 2 3 4 5 6 7 8 9 10 11 12 13
BFS: Analysis, I

Global initialization: mark all vertices "undiscovered"

BFS(s)

- mark s "discovered"
- queue = { s }

while queue not empty

- u = remove_first(queue)

for each edge {u,x}

- if (x is undiscovered)
  - mark x discovered
  - append x on queue

- mark u fully explored

Simple analysis:
2 nested loops.
Get worst-case number of iterations of each; multiply.

O(n) + O(1) + O(n) x O(n) = O(n^2)
BFS: Analysis, II

Above analysis correct, but pessimistic, assuming G is sparse, edge list representation: can't have $\Omega(n)$ edges incident to each of $\Omega(n)$ distinct "u" vertices.

Alt, more global analysis:

Each edge is explored once from each end-point, so total runtime of inner loop is $O(m)$, (assuming edge-lists)

Total $O(n+m)$, $n = \# \text{ nodes}$, $m = \# \text{ edges}$

Exercise: extend algorithm and analysis to non-connected graph

Bipartiteness
Properties of (Undirected) BFS(v)

BFS(v) visits x if and only if there is a path in G from v to x.

Edges into then-undiscovered vertices define a tree – the "breadth first spanning tree" of G

Level i in this tree are exactly those vertices u such that the shortest path (in G, not just the tree) from the root v is of length i.

All non-tree edges join vertices on the same or adjacent levels
BFS Application: Shortest Paths

Tree (solid edges) gives shortest paths from start vertex.

Can label by distances from start, all edges connect same/adjacent levels.
BFS Application: Shortest Paths

*Tree* (solid edges) gives shortest paths from start vertex.

- Can label by distances from start.
- All edges connect same/adjacent levels.
BFS Application: Shortest Paths

Tree (solid edges) gives shortest paths from start vertex.

Can label by distances from start. All edges connect same/adjacent levels.
BFS Application: Shortest Paths

*Tree* (solid edges) gives shortest paths from start vertex

- Can label by distances from start
- All edges connect same/adjacent levels
Why fuss about trees?

Trees are simpler than graphs
Ditto for algorithms on trees vs algs on graphs
So, this is often a good way to approach a graph problem: find a "nice" tree in the graph, i.e., one such that non-tree edges have some simplifying structure
E.g., BFS finds a tree s.t. level-jumps are minimized
DFS (below) finds a different tree, but it also has interesting structure…
Graph Search Application: Connected Components

Want to answer questions of the form:
Given vertices u and v, is there a path from u to v?

Idea: create array A such that
A[u] = smallest numbered vertex that is connected to u. Question reduces to whether A[u]=A[v]?

Q: Why not use 2-d array Path[u,v]?

Graph Search Application: Connected Components

initial state: all $v$ undiscovered
for $v = 1$ to $n$ do
  if state($v$) == undiscovered then
    BFS($v$): setting $A[u] \leftarrow v$ for each $u$ found
    (and marking $u$ discovered)
  endif
endfor

Total cost: $O(n+m)$  Naively, three nested loops $\Rightarrow O(n^3)$, but careful look at BFS($v$) shows $O(n_i+m_i)$ if $v$’s component has $n_i$ nodes & $m_i$ edges; $\Sigma n_i+m_i = n+m$. Idea: each edge is touched twice, once from each end. (True for DFS, too)
3.4 Testing Bipartiteness
Bipartite Graphs

Def. An undirected graph $G = (V, E)$ is **bipartite** (2-colorable) if the nodes can be colored red or blue such that no edge has both ends the same color.

Applications.
- Stable marriage: men = red, women = blue
- Scheduling: machines = red, jobs = blue

"bi-partite" means "two parts." An equivalent definition: $G$ is bipartite if you can partition the node set into 2 parts (say, blue/red or left/right) so that all edges join nodes in different parts/no edge has both ends in the same part.

*a bipartite graph*
Testing Bipartiteness

**Testing bipartiteness.** Given a graph $G$, is it bipartite?

Many graph problems become:
- easier if the underlying graph is bipartite (matching)
- tractable if the underlying graph is bipartite (independent set)

Before attempting to design an algorithm, we need to understand structure of bipartite graphs.

*a bipartite graph* $G$

*another drawing of* $G$
Lemma. If a graph $G$ is bipartite, it cannot contain an odd length cycle.

Pf. Impossible to 2-color the odd cycle, let alone $G$. 
Bipartite Graphs

Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.

(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Case (i)

Case (ii)
Bipartite Graphs

Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (i)
Suppose no edge joins two nodes in the same layer.
By previous lemma, all edges join nodes on adjacent levels.

Bipartition: red = nodes on odd levels, blue = nodes on even levels.

Case (i)
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (ii) Suppose $(x, y)$ is an edge & $x, y$ in same level $L_j$. Let $z = \text{their lowest common ancestor in BFS tree}$. Let $L_i$ be level containing $z$. Consider cycle that takes edge from $x$ to $y$, then tree from $y$ to $z$, then tree from $z$ to $x$. Its length is $1 + (j-i) + (j-i)$, which is odd.
Obstruction to Bipartiteness

Cor: A graph $G$ is bipartite iff it contains no odd length cycle.

NB: the proof is algorithmic—it finds a coloring or odd cycle.
3.6 DAGs and Topological Ordering

This should be review of 331/373 material

I won’t lecture on it, but you should read book/slides to be sure it makes sense, with emphasis on correctness, analysis.
Precedence Constraints

Precedence constraints. Edge \((v_i, v_j)\) means task \(v_i\) must occur before \(v_j\).

Many Applications

Course prerequisites: course \(v_i\) must be taken before \(v_j\)

Compilation: must compile module \(v_i\) before \(v_j\)

Computing workflow: output of job \(v_i\) is input to job \(v_j\)

Manufacturing or assembly: sand it before you paint it…

Spreadsheet evaluation order: if A7 is "=A6+A5+A4", evaluate 4,5,6 first
Directed Acyclic Graphs

Def. A **DAG** is a directed acyclic graph, i.e., one that contains no directed cycles.

Ex. Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

Def. A **topological order** of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).

E.g., \(\forall\) edge \((v_i, v_j)\), finish \(v_i\) before starting \(v_j\)

![a DAG](image1)

![a topological ordering of that DAG— all edges oriented left-to-right](image2)
Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Pf. (by contradiction)

Suppose that G has a topological order \(v_1, \ldots, v_n\) and that G also has a directed cycle C.
Let \(v_i\) be the lowest-indexed node in C, and let \(v_j\) be the node just before \(v_i\); thus \((v_j, v_i)\) is an edge.
By our choice of \(i\), we have \(i < j\).
On the other hand, since \((v_j, v_i)\) is an edge and \(v_1, \ldots, v_n\) is a topological order, we must have \(j < i\), a contradiction.

if all edges go L→R, you can't loop back to close a cycle
Directed Acyclic Graphs

Lemma (above).
If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?
Q. If so, how do we compute one?
Lemma. If $G$ is a DAG, then $G$ has a node with no incoming edges.

Pf. (by contradiction)
Suppose that $G$ is a DAG and every node has at least one incoming edge. Let's see what happens.
Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$. Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.
Repeat until we visit a node, say $w$, twice. Let $C$ be the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle, contradicting acyclicity.

Why must this happen?
Directed Acyclic Graphs

Lemma. If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)

Base case: true if $n = 1$.

Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.

$G - \{ v \}$ is a DAG, since deleting $v$ cannot create cycles.

By inductive hypothesis, $G - \{ v \}$ has a topological ordering.

Place $v$ first in topological ordering; then append nodes of $G - \{ v \}$ in topological order. This is valid since $v$ has no incoming edges. 

\[ \blacksquare \]

To compute a topological ordering of $G$:

Find a node $v$ with no incoming edges and order it first

Delete $v$ from $G$

Recursively compute a topological ordering of $G - \{v\}$

and append this order after $v$
Topological Ordering Algorithm: Example

Topological order:
Topological Order: $v_1$
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2$
Topological Ordering Algorithm: Example

Topological order: \( v_1, v_2, v_3 \)
Topological Ordering Algorithm: Example

Topological order: \(v_1, v_2, v_3, v_4\)
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5$
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$
Topological Ordering Algorithm: Example

Topological order: \(v_1, v_2, v_3, v_4, v_5, v_6, v_7\).
Topological Sorting Algorithm

Maintain the following:
- \( \text{count}[w] = \) (remaining) number of incoming edges to node \( w \)
- \( S = \) set of (remaining) nodes with no incoming edges

Initialization:
- \( \text{count}[w] = 0 \) for all \( w \)
- \( \text{count}[w]++ \) for all edges \((v,w)\)
- \( S = S \cup \{w\} \) for all \( w \) with \( \text{count}[w] == 0 \)

Main loop:
- while \( S \) not empty
  - remove some \( v \) from \( S \)
  - make \( v \) next in topo order
  - for all edges from \( v \) to some \( w \)
    - \( \text{count}[w]-- \)
    - if \( \text{count}[w] == 0 \) then add \( w \) to \( S \)

Correctness: clear, I hope

Time: \( O(m + n) \) (assuming edge-list representation of graph)

why does it terminate?

what if \( G \) has cycle?

nested loops: why not \( n \cdot m \)?
Depth-First Search
Depth-First Search

Follow the first path you find as far as you can go
When you reach a dead end, back up to last unexplored edge, then go as far you can. Etc.

Naturally implemented using recursion or a stack
DFS(v) – Recursive version

Global Initialization:

for all nodes v, v.dfs# = -1 // mark v "undiscovered"
dfscounter = 0

DFS(v):

v.dfs# = dfscounter++ // v "discovered", number it
for each edge (v,x)
  if (x.dfs# = -1) // tree edge (x previously undiscovered)
    DFS(x)
  else … // code for back-, fwd-, parent-
// edges, if needed; mark v
// "completed," if needed
Why fuss about trees (again)?

BFS tree ≠ DFS tree, but, as with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple" – only descendant/ancestor

Proof below
Suppose edge lists at each vertex are sorted alphabetically.
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A, B)

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**

Call Stack:
- (Edge list)
- A (B, J)
- B (A, C, J)
Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C)
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D)

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - D (C,E,F)

Color code:
- undiscovered
- discovered
- fully-explored
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

First Traversal: (A,B) (B,C) (C,D) (D,E)

**Call Stack:**
- (A, B, J)
- (A, C, J)
- (B, D, G, H)
- (C, E, F)
- (D, F)

**Color code:**
- **undiscovered**
- **discovered**
- **fully-explored**
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A, B) (B, C) (C, D) (D, E) (E, F)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:

- (Edge list)
  - A (B, J)
  - B (A, C, J)
  - C (B, D, G, H)
  - D (C, E, F)
  - E (D, F)
  - F (D, E, G)
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(Edge list)

A (B, J)
B (A, C, J)
C (B, D, G, H)
D (C, E, F)
E (D, F)
F (D, E, G)
G (C, F)

First Traversal: (A, B) (B, C) (C, D) (D, E) (E, F) (F, D) (F, G)
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A, B) (B, C) (C, D) (D, E) (E, F) (F, D) (F, G) (G, C)

Call Stack:
- A (B, J)
- B (A, C, J)
- C (B, D, G, H)
- D (G, E, F)
- E (D, F)
- F (D, E, G)
- G (C, F)

Color code:
- undiscovered
- discovered
- fully-explored
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,G) (G,C)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - D (C,E,F)
  - E (D,F)
  - F (D,E,G)
  - G (C)

Diagram:
- Vertices labeled with positions (e.g., A, 1)
- Edges connecting vertices
- Color coding for vertices and edges
Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - D (C,E,F)
  - E (D,F)

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C)
Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,G) (G,C)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - (A,B,J)
  - (A,C,J)
  - (B,D,G,H)
  - (C,E,F)

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,G) (G,C)
Suppose edge lists at each vertex are sorted alphabetically.

DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B, J)
  - B (A, C, J)
  - C (B, D, G, H)

First Traversal: (A, B) (B, C) (C, D) (D, E) (E, F) (F, D) (F, G) (G, C)
Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H)

Color code: undiscovered discovered fully-explored
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,G) (G,C) (C,H) (H,I)

Call Stack: (Edge list)
- A (B,J)
- B (A,C,J)
- C (B,D,G,H)
- H (C,I,J)
- I (H)

Color code:
- undiscovered
- discovered
- fully-explored

**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,G) (G,C) (C,H) (H,I)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - H (C,I,J)
  - I (H)

- (Edge list)
  - K
  - L
  - M
Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,G) (G,C) (C,H) (H,I)
Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J)
Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J) (J,A) (J,B) (J,K)
Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J) (J,A) (J,B) (J,K) (K,L)
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J) (J,A) (J,B) (J,K) (K,L) (L,J) (L,M)

Color code:
- undiscovered
- discovered
- fully-explored
Suppose edge lists at each vertex are sorted alphabetically.
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J) (J,A) (J,B) (J,K) (K,L) (L,J) (L,M)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - H (C,I,J)
  - J (A,B,H,K,L)
  - K (J,L)

```plaintext
DFS(A)
```

```plaintext
Suppose edge lists at each vertex are sorted alphabetically.
```

```plaintext
First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J) (J,A) (J,B) (J,K) (K,L) (L,J) (L,M)
```

```plaintext
Color code:
- undiscovered
- discovered
- fully-explored
```
Suppose edge lists at each vertex are sorted alphabetically.
Suppose edge lists at each vertex are sorted alphabetically.
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J) (J,A) (J,B) (J,K) (K,L) (L,J) (L,M)
Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J) (J,A) (J,B) (J,K) (K,L) (L,J) (L,M)
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (I,J) (J,A) (J,B) (J,K) (K,L) (L,J) (L,M)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)

**DFS(A)**

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (I,J) (J,A) (J,B) (J,K) (K,L) (L,J) (L,M)
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(Edge list)

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J) (J,A) (J,B) (J,K) (K,L) (L,J) (L,M)
Suppose edge lists at each vertex are sorted alphabetically.

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J) (J,A) (J,B) (J,K) (K,L) (L,J) (L,M)

Call Stack:
(Edge list)
A (B,C)
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J) (J,A) (J,B) (J,K) (K,L) (L,J) (L,M)

Color code: undiscovered discovered fully-explored

Call Stack: (Edge list) TA-DA!!
DFS(A)

Edge code:
- Tree edge
- Back edge

Diagram:
- A,1
- B,2
- C,3
- D,4
- E,5
- F,6
- G,7
- H,8
- I,9
- J,10
- K,11
- L,12
- M,13

Tree edges are shown as solid lines, and back edges are shown as dotted lines.
DFS(A)

Edge code:
- Tree edge
- Back edge
DFS(A)

Edge code:
- Tree edge
- Back edge
DFS(A)

Edge code:
Tree edge
Back edge

A,1
B,2
C,3
D,4
E,5
F,6
G,7
H,8
I,9
J,10
K,11
L,12
M,13
DFS(A)

Edge code:
- Tree edge
- Back edge
DFS(A)

Edge code:
- Tree edge
- Back edge
DFS(A)

Edge code:
- Tree edge
- Back edge
- No Cross Edges!
Properties of (Undirected) DFS(v)

Like BFS(v):

DFS(v) visits x if and only if there is a path in G from v to x (through previously unvisited vertices)

Edges into then-undiscovered vertices define a tree – the "depth first spanning tree" of G

Unlike the BFS tree:

the DF spanning tree isn't minimum depth
its levels don't reflect min distance from the root
non-tree edges never join vertices on the same or adjacent levels

BUT…
Non-tree edges

All non-tree edges join a vertex and one of its descendents/ancestors in the DFS tree

No cross edges!
Why fuss about trees (again)?

As with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple" – only descendant/ancestor
A simple problem on trees

**Given:** tree $T$, a value $L(v)$ defined for every vertex $v$ in $T$

**Goal:** find $M(v)$, the min value of $L(u)$ for any $u$ in the subtree rooted at $v$ (including $v$ itself).

**How?** Depth first search, using:

$$M(v) = \begin{cases} L(v) & \text{if } v \text{ is a leaf} \\ \min(L(v), \min_{w \text{ a child of } v} M(w)) & \text{otherwise} \end{cases}$$
Application: Articulation Points

A node in an undirected graph is an articulation point iff removing it disconnects the graph (or, more generally, increases the number of connected components)

Articulation points represent, e.g.:
- vulnerabilities in a network – single points whose failure would split the network into 2 or more disconnected components
- bottlenecks to information flow in a network

...
Identifying key proteins on the anthrax predicted network

Articulation point proteins
Articulation Points

articulation point
iff its removal disconnects the graph
Articulation Points
Simple Case: Artic. Pts in a tree

Leaves – never articulation points
Internal nodes – always articulation points
Root – articulation point if and only if it has two or more children

Non-tree: extra edges remove some articulation points (which ones?)
Articulation Points
DFS(A)

Edge code:
- Tree edge
- Back edge
- No Cross Edges!

△ Articulation points
Articulation Points from DFS

Root node is an articulation point iff it has more than one child
Leaf is never an articulation point

Non-leaf, non-root node $u$ is an articulation point

$\exists$ some child $y$ of $u$ s.t. no non-tree edge goes above $u$ from $y$ or below

If $u$’s removal does NOT separate $x$, there must be an exit from $x$’s subtree. How? Via back edge.
Articulation Points: the "LOW" function

Definition: LOW(v) is the lowest dfs# of any vertex that is either in the dfs subtree rooted at v (including v itself) or directly connected to a vertex in that subtree by one back edge.

Key idea 1: if some child x of v has \(\text{LOW}(x) \geq \text{dfs#}(v)\) then v is an articulation point (excl. root)

Key idea 2: \(\text{LOW}(v) = \min\left(\{\text{dfs#}(v)\} \cup \{\text{LOW}(w) \mid w \text{ a child of } v\} \cup \{\text{dfs#}(x) \mid \{v,x\} \text{ is a back edge from } v\}\right)\)
DFS To Find Articulation Points

Global initialization: dfscounter = 0; v.dfs# = -1 for all v.

DFS(v):
  v.dfs# = dfscounter++
  v.low = v.dfs#  // initialization
for each edge {v,x}
  if (x.dfs# == -1)  // x is undiscovered
    DFS(x)
  v.low = min(v.low, x.low)
  if (x.low >= v.dfs#)
    print "v is art. pt., separating x"
else if (x is not v's parent)
  v.low = min(v.low, x.dfs#)

Equiv: "if( {v,x} is a back edge)"
Why?

Except for root. Why?
What if G is not connected?
Articulation Point

LOW(v) = highest exit from v’s subtree
Articulation Points

LOW(v) = highest exit from v’s subtree

<table>
<thead>
<tr>
<th>Vertex</th>
<th>DFS #</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>H</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>J</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>K</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>L</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>M</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>
Summary

Graphs – abstract relationships among pairs of objects
Terminology – node/vertex/vertices, edges, paths, multi-edges, self-loops, connected
Terminology (trees) – root, leaf, parent, child, sibling, …
Representation – edge list, adjacency matrix
Nodes vs Edges – $m = O(n^2)$, often less (sparse/dense)
BFS – Layers, queue, shortest paths, all edges go to same or adjacent layer, tree, global analysis of nested loops
DFS – recursion/stack; all edges ancestor/descendant
Algorithms – connected components, shortest path, bipartiteness, topological sort, articulation points