Objects & Relationships

- The Kevin Bacon Game:
  - Actors
  - Two are related if they've been in a movie together
- Exam Scheduling:
  - Classes
  - Two are related if they have students in common
- Traveling Salesperson Problem:
  - Cities
  - Two are related if can travel directly between them

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
Undirected Graph $G = (V,E)$

1. 2
2. 3
3. 4
4. 5
5. 6
6. 7
7. 8
8. 9
9. 10
10. 11
11. 12
12. 13
13. 14

loop

multi-edge
Undirected Graph $G = (V,E)$

Directed Graph $G = (V,E)$

Graphs don’t live in Flatland

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

Specifying undirected graphs as input

- What are the vertices?
  - 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:

  \[
  \begin{array}{cccc}
  A & 7 & 3 & 4 \\
  A & 0 & 0 & 1 & 1 \\
  7 & 0 & 0 & 0 & 1 \\
  3 & 1 & 0 & 0 & 1 \\
  4 & 1 & 1 & 1 & 0 \\
  \end{array}
  \]
Specifying directed graphs as input

- What are the vertices
  - 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>1</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 \textit{different} vertices (no loops), and no two edges connect the \textit{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 \textit{sparse} if \(m \ll n^2\), otherwise it is \textit{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
- 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 \(G = (V,E)\) \(n\) vertices, \(m\) edges

- 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.
  - compact if in packed binary form for large \(m\)
- Disadvantages: inefficient for sparse graphs
  - \(m \ll n^2\)
Representing Graph $G=\langle V, E \rangle$

$n$ 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

Graph Traversal

- Learn the basic structure of a graph
- “Walk,” via edges, from a fixed starting vertex $v$ to all vertices reachable from $v$

- Three states of vertices
  - undiscovered
  - discovered
  - fully-explored

Breadth-First Search

- Completely explore the vertices in order of their distance from $v$

- Naturally implemented using a queue
**BFS(v)**

Global initialization: mark all vertices "undiscovered"
BFS(v)
  mark v "discovered"
  queue = v
  while queue not empty
    u = remove_first(queue)
    for each edge (u,v)
      if (v is undiscovered)
        mark v discovered
        append v on queue
    mark u completed

Exercise: modify code to number vertices & compute level numbers
BFS analysis

- Each edge is explored once from each end-point (at most)
- Each vertex is discovered by following a different edge
- Total cost $O(m)$ where $m$=# of edges

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

All edges connect same/adjacent levels
Why fuss about trees?

- Trees are simpler than graphs
- Ditto for algorithms on trees vs 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 (next) 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 create 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) != fully-explored then
      BFS(v): setting A[u] ← v for each u found (and marking u discovered/fully-explored)
    endif
  endfor
- Total cost: O(n+m)
  - each edge is touched a constant number of times
  - works also with DFS

Depth-First Search

- Follow the first path you find as far as you can go
- Back up to last unexplored edge when you reach a dead end, then go as far you can
- Naturally implemented using recursive calls or a stack
DFS(v) - explicit stack

Global Initialization: mark all vertices "undiscovered"  
DFS(v)
mark v "discovered"  
push (v,1) onto empty stack  
while stack not empty
(u,i) = pop(stack)
for ( ; i ≤ # of neighbors of u; i++)
x = ith edge on u’s edge list  
if (x is undiscovered)
mark x "discovered"  
push (u,i+1)  // save info to resume with u's next edge,
u = x  // after exploring from x,  
i = 1  // (starting with its first edge)
mark u completed

Exercise: modify to compute vertex numbering

DFS(v) – Recursive version

Global Initialization: mark all vertices v "undiscovered" via v.dfs# = -1  
dfscounter = 0
DFS(v)
v.dfs# = dfscounter++  // mark 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

Suppose edge lists at each vertex are sorted alphabetically

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

Suppose edge lists at each vertex are sorted alphabetically

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

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
Suppose edge lists at each vertex are sorted alphabetically.

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

**Call Stack:**
- (Edge list)

DFS(A)

A,1

B,2

J

C,3

G,7

H

K

L

I

M

D,4

F,6

E,5

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

DFS(A)

A,1

B,2

J

C,3

G,7

H

K

L

I

M

D,4

F,6

E,5

Call Stack:
- (Edge list)
  - A (J)
  - B (F, J)
  - C (F, G, H)
  - D (G, F)
  - E (F)
  - F (G, E, P)
  - G (C, F)
Suppose edge lists at each vertex are sorted alphabetically.

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

**Call Stack:**
(Edge list)

**DFS(A)**
- A,1
- B,2
- C,3
- D,4
- E,5
- F,6
- G,7
- H
- I
- J
- K
- L
- M

55

56

57

58
Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)

DFS(A)

A,1
B,2
C,3
D,4
E,5
F,6
G,7
H,8
I,9
J
K
L
M

15
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

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

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

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

**DFS(A)**

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

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

Call Stack:
- **(Edge list)**
- A (F, J)
- B (F, J)
- C (F, J)
- D (F, J)
- E (F, J)
- F (J, L)
- G (J, L)
- H (J, L)
- I (J, L)
- J (J, L)
- K (J, L)
- L (J, K, M)
- M (J, K, M)
Suppose edge lists at each vertex are sorted alphabetically.

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

**Call Stack:** (Edge list)

A (F, J)
B (J, G, J)
C (G, F, G, J)
H (J, G, H)
J (J, F, K, J)
K (J, K, J)
L (J, K, L)
M (J, K, L)

1. DFS(A)
   - 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

2. DFS(A)
   - 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

3. DFS(A)
   - 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

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

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>undiscovered</td>
</tr>
<tr>
<td>B</td>
<td>discovered</td>
</tr>
<tr>
<td>C</td>
<td>fully-explored</td>
</tr>
</tbody>
</table>

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

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

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
- A

Edge code:
- Tree edge
- Back edge

DFS(A)

Edge code:
- Tree edge
- Back edge

DFS(A)
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 $L(v)$ anywhere 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.

- Articulation points represent vulnerabilities in a network – single points whose failure would split the network into 2 or more disconnected components.

Identifying key proteins on the anthrax predicted network

Articulation point proteins

Ram Samudrala/Jason McDermott
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 two or more children
- Non-tree: extra edges remove some articulation points (which ones?)

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 removal of $u$ 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 connected to a vertex in that subtree by a back edge.
- Key idea 1: if some child $x$ of $v$ has $LOW(x) \geq dfs#(v)$ then $v$ is an articulation point (excl. root)
- Key idea 2: $LOW(v) = \min \{ \{dfs#(v)\} \cup \{LOW(w) \mid w \text{ a child of } v\} \cup \{dfs#(x) \mid \{v,x\} \text{ is a back edge from } v\} \}$
**DFS(v) for Finding Articulation Points**

Global initialization: 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?*

**Articulation Points**

<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>5</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>H</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>I</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>J</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>K</td>
<td>11</td>
<td>10</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>

AP's: BCC’s: 1) 2) 3) 4)
Articulation Points

<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>3</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>H</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

AP's: C, B, F

BCC's:
1) C--D, D--E, E--C
2) B--C
3) A--B, B--F, F--A
4) F--G, G--H, H--F