Representations of Graphs

- OR -

String[] names;
boolean[][] edgeExists;

class Vertex
{
    String name;
    Vertex[] links;
}

We'll show you a third, as well! But no space on the slide...
Implementing a Graph in code

• Q. How to represent a graph?
  – Set of vertices
  – Set of edges
  – Weight function (if weighted)

• A. Try an *adjacency list* or *adjacency matrix*.

• All the coding concepts in this section are flexible, and you should adapt them as needed.
  – Bigger projects may need fancier designs;
  – Simple projects may not need this much complexity.
Adjacency list (basic)

- An array, indexed by vertices, storing a list of adjacent vertices. Each row is a vertex object.

<table>
<thead>
<tr>
<th>VERTEX</th>
<th>. . . IS ADJACENT TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 3, 4, 5</td>
</tr>
<tr>
<td>1</td>
<td>0, 2, 4</td>
</tr>
<tr>
<td>2</td>
<td>3, 1</td>
</tr>
<tr>
<td>3</td>
<td>0, 2, 4</td>
</tr>
<tr>
<td>4</td>
<td>0, 1, 3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

![Diagram of a graph with vertices labeled from 0 to 5 and edges connecting them]
If it's a graph:
Each edge is represented in both directions

D, E, B all know about their edges “to” A.
A knows about all three of its edges “to” the others.
In a graph, all edges are symmetric.
Adjacency list (via an edge list)

- Array indexed by vertices, each storing a list of incident edges.
- Each edge stores its incident vertices.

<table>
<thead>
<tr>
<th>VTX.</th>
<th>INCIDENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e0, e1, e2, e3</td>
</tr>
<tr>
<td>1</td>
<td>e2, e4</td>
</tr>
<tr>
<td>2</td>
<td>e6</td>
</tr>
<tr>
<td>3</td>
<td>e1, e5, e6</td>
</tr>
<tr>
<td>4</td>
<td>e3, e4, e5</td>
</tr>
<tr>
<td>5</td>
<td>e0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EDGE</th>
<th>INCIDENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0</td>
<td>0, 5</td>
</tr>
<tr>
<td>e1</td>
<td>0, 3</td>
</tr>
<tr>
<td>e2</td>
<td>0, 1</td>
</tr>
<tr>
<td>e3</td>
<td>0, 4</td>
</tr>
<tr>
<td>e4</td>
<td>1, 4</td>
</tr>
<tr>
<td>e5</td>
<td>3, 4</td>
</tr>
<tr>
<td>e6</td>
<td>2, 3</td>
</tr>
<tr>
<td>e7</td>
<td>1, 2</td>
</tr>
</tbody>
</table>

Edge list, a/k/a “incidence container”
Representing Edges

**Adjacency Matrix:**

$|V| \times |V|$ matrix of booleans

(or numbers, for a weighted graph)
Representing Edges

Adjacency Matrix:
For undirected graphs, the matrix always is symmetric -
(the matrix equals its transpose).
We can save space if desired.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>-</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>
Representing Edges

**Adjacency Matrix:**
For digraphs, the matrix is not in general symmetric.

\[
\begin{array}{cccc}
A & B & C & D \\
A & 0 & 1 & 0 & 0 \\
B & 0 & 0 & 0 & 0 \\
C & 1 & 0 & 0 & 0 \\
D & 0 & 0 & 1 & 0 \\
\end{array}
\]
Matrix vs. List

- **Adjacency List:**
  - More space efficient for *sparse* graphs, that is, graphs with $|E| << |V|^2$.
  - $O(|E|)$ time to test adjacency of two vertices.

- **Adjacency Matrix:**
  - $O(1)$ access time to test adjacency of two vertices.
  - Wastes a lot of space for a very sparse graph.
  - Space efficient if graph is *dense* (many edges) -- one bit per edge in an unweighted graph.
Simple implementation of Adjacency List (implicit edges)

```java
public class Vertex
{
    List<Vertex> neighbors;
    Object       label_info;
}

Vertex[] allVertices;
```

Each Vertex carries around its own list of neighbors.
Simple implementation of Adjacency List with Edge List

```java
public class Vertex {
    List<Edge> edges;
    Object label_info;
}

public class Edge {
    Vertex v1, v2;
    Object label_info;
}

Vertex[] allVertices;
```

Each Vertex carries references to its incident edges.
public class Vertex
{
    int index;
    Object data;
}

Vertex [] allVertices;
boolean[][][] adjacencyMatrix;

Maybe you could eliminate the Vertex class altogether???
Simple Implementation Summary

• All those simple implementations are just sketches that you should customize appropriately:
  – Perhaps use an ArrayList or some kind of Map in the Vertex.
  – Customize the label information to meet your requirements.
BFS and DFS

Sources:
https://scratch.mit.edu/projects/103492660/
http://computer.howstuffworks.com/routing-algorithm5.htm
Google Maps
**BFS and DFS**

**How to Traverse a Graph?**

- Some algorithms need to see every vertex
- Some just “hunt until answer is found”
- It's not as simple as a tree traversal:
  - Due to Cycles, we cannot just follow every link now.
  - We need to deal with redundant paths,
  - so we add a “isVisited” flag to each vertex.
BFS and DFS

- **DFS: Depth First Search**
  - Like preorder traversal in a rooted tree.
  - Start at a vertex, find any neighbor.
  - Visit it, and recurse from there.
  - After recursion is done, recurse into next neighbor.

- **BFS: Breadth First Search**
  - Start at some vertex, find all (adjacent) neighbors.
  - Visit each neighbor,
  - . . . and only then visit their neighbors, and so on.
DFS pseudocode

for i ← 0 to allVertices.length-1
    allVertices[i].visited ← false

for i ← 0 to allVertices.length-1
    if not allVertices[i].visited
        DFS(allVertices[i])

DFS(v)
    /* “Visit” this vertex -- do whatever you need to with it.
    * Print, compute, label, who knows.
    */
    v.visited ← true

    // Recurse on unvisited neighbors.
    for each w in v.neighbors
        if not w.visited
            DFS(w)
DFS in Trees

Now for DFS.
We'll start at D.
We recurse into an arbitrary neighbor, B.

NOTE: Red vertices represent the stack.
We recurse again, into a neighbor of B.
We return to B, and then recurse into another neighbor of B.
We return to A, and then explore the other subtree.
We return to A, and then explore the other subtree.
DFS in Trees

We return to A, and then explore the other subtree.
PREPARING FOR A DATE:

WHAT SITUATIONS MIGHT I PREPARE FOR?
1) MEDICAL EMERGENCY
2) DANCING
3) FOOD TOO EXPENSIVE

OKAY, WHAT KINDS OF EMERGENCIES CAN HAPPEN?
A) SNAKEBITE
B) LIGHTNING STRIKE
C) FALL FROM CHAIR
D) DANCEFLOOR DROWNING

HMM. WHICH SNAKES ARE DANGEROUS? LET'S SEE...
1) A) CORN SNAKE
B) GARTER SNAKE
C) COPPERHEAD

THE RESEARCH COMPARING SNAKE VENOMS IS SCATTERED AND INCONSISTENT. I'LL MAKE A SPREADSHEET TO ORGANIZE IT.

I'M HERE TO PICK YOU UP. YOU'RE NOT DRESSED?
BY LP, THE INLAND TAIPAN HAS THE DEADIEST VENOM OF ANY SNAKE!

I REALLY NEED TO STOP USING DEPTH-FIRST SEARCHES.
Time Complexity of DFS

- First loop to clear flags: $\Theta(|V|)$.
- 2nd loop to launch DFS at each vertex:
  - $\Theta(|V|) + \text{total cost of all DFS calls}$.
- Cost of all DFS calls (considered together):
  - DFS is called exactly once for each vertex in $V$.
  - In DFS($v$), the for-each loop runs $\text{degree}(v)$ times.
  - Total number of those tests (in all calls): $2|E|$.
- Total cost: $\Theta(|V| + |E|)$
  - For most graphs, that means the cost is $\Theta(|E|)$. 
What is a graph traversal good for?

• Printing every vertex.
• Finding connected components, and labeling each vertex with a component number.
• Finding *spanning trees*.
• Finding a path from any vertex to any other reachable vertex.
• Testing for cycles.
• And more!
Finding connected components with DFS

for i ← 0 to allVertices.length-1
    allVertices[i].component ← -1 // -1 means it is unlabeled

current_component_num ← 0

for i ← 0 to allVertices.length-1
    if allVertices[i].component < 0 // if vertex is unlabeled
        DFS2(allVertices[i], current_component_num++)

print("# of connected components: " + current_component_num)

DFS2(v, ccn)
    // Label this vertex with connected component number ccn.
    v.component ← ccn

    // Recurse on unlabeled neighbors.
    for each w in v.neighbors
        if w.component < 0 // if vertex is unlabeled
            DFS2(w, ccn)
Finding a path from \( x \) to \( y \) with DFS

- First, label vertices with connected component number. If \( x \) and \( y \) are in different connected components, give up: no path exists. Else,

```java
for i ← 0 to allVertices.length-1
    allVertices[i].visited ← false
    allVertices[i].parent ← null
DFS3(y) // Launch at y, building a DFS tree that will find x.
v ← x
while v ≠ y
    print("Path vertex: " + v)
    v ← v.parent
DFS3(v)
```

```java
v.visited ← true // Search from v.
for each w in v.neighbors // Visit the unvisited
    if not w.visited
        w.parent ← v // Store link toward root
        DFS3(w)
```

93
What are these parent pointers?

- Parent pointers (from a DFS search) define a spanning tree in an undirected graph.
  - Child points to parent, but not vice versa.
  - Unused edges (back edges) connect ancestor and descendant.
Wolf, Goat, Cabbage puzzle

• You have a wolf, a goat, a cabbage.
  – The wolf wants to eat the goat, so don't leave them alone together.
  – The goat wants to eat the cabbage, so don't leave them alone together.

• You have to cross a river, and the boat can only hold you and one other thing.
  – You want all three things with you on the other side.
  – You will have to take multiple trips.
Let's simulate BFS.
We'll start at D.
BFS in Trees

We mark D as "done" - we've inspected it.
We move on to the neighbors of D – B, F.
We move on to the neighbors of D – B, F.
Now, we go to 2nd order neighbors (neighbors of B,F).
Now, we go to $2^{nd}$ order neighbors (neighbors of B, F).
BFS in Trees

Now, we go to 2\textsuperscript{nd} order neighbors (neighbors of B,F).
Now, we go to 2nd order neighbors (neighbors of B,F).
BFS pseudocode

for i ← 0 to allVertices.length-1
    allVertices[i].visited ← false
    allVertices[i].parent ← null

for i ← 0 to allVertices.length-1
    if not allVertices[i].visited
        BFS(allVertices[i])

BFS(v)
    Queue<Vertex> q  // Initialize FIFO of vertices
    v.visited ← true  // Visit v
    q.add(v)
    while q is not empty,
        w ← q.remove()  // Dequeue next subtree-root
        for each x in w.neighbors
            if not x.visited
                x.visited ← true // Visit x, then enqueue
                x.parent ← w
                q.add(x)
BFS vs. DFS

Should I use BFS or DFS?

- Depends on what kind of spanning tree you want
- DFS tree is tall, thin.
  - Recursive code is so simple though.
- BFS tree is wider, shorter.
  - Minimum number of edges to root
Detecting Cycles

• What's a situation where you might care about cycles?

• Finding cycles with DFS, check for back edges.
  – The graph has a cycle iff DFS finds a back edge.

• How do you detect a back edge?
  – When examining neighbors of v, if a neighbor w is already visited, AND, w is not v's DFS parent, then edge {v, w} is a back edge.

• Also you can use BFS, and detect cross edges.
Finding Cycles with DFS

DFS4(v)

v.visited ← true

// Recurse on unvisited neighbors.
for each w in v.neighbors
    if not w.visited
        // Store link toward root
        w.parent ← v
        DFS4(w)
    else if w ≠ v.parent
        // Back edge is {v,w}
        print(“Cycle found”)
Detecting Cycles

Let's look for cycles using DFS.

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>done</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
Detecting Cycles
Detecting Cycles

Note that at this point, A checks its neighbors and tries to recurse. Suppose it tries E first.

We notice that E is already visited, so we do not recurse, but we note *back edge* \{A,E\}.
Instead, we recurse to D, then C.
Detecting Cycles

Our stack at this point is:

- C
- D
- A
- B
- E

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>done</td>
</tr>
<tr>
<td>B</td>
<td>done</td>
</tr>
<tr>
<td>C</td>
<td>done</td>
</tr>
<tr>
<td>D</td>
<td>done</td>
</tr>
<tr>
<td>E</td>
<td>done</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
Detecting Cycles

We return to A, and the stack is:
- A
- B
- E

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>done</td>
</tr>
<tr>
<td>B</td>
<td>done</td>
</tr>
<tr>
<td>C</td>
<td>done</td>
</tr>
<tr>
<td>D</td>
<td>done</td>
</tr>
<tr>
<td>E</td>
<td>done</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
Detecting Cycles

We recurse into another child of A:

- F
- A
- B
- E

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>done</td>
</tr>
<tr>
<td>B</td>
<td>done</td>
</tr>
<tr>
<td>C</td>
<td>done</td>
</tr>
<tr>
<td>D</td>
<td>done</td>
</tr>
<tr>
<td>E</td>
<td>done</td>
</tr>
<tr>
<td>F</td>
<td>done</td>
</tr>
</tbody>
</table>
Detecting Cycles

- When we finish, every edge is labeled as a \textit{tree edge} (in the DFS spanning tree), or a \textit{back edge} (to an ancestor of the parent).

Graph is acyclic iff there are no back edges.

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>done</td>
</tr>
<tr>
<td>B</td>
<td>done</td>
</tr>
<tr>
<td>C</td>
<td>done</td>
</tr>
<tr>
<td>D</td>
<td>done</td>
</tr>
<tr>
<td>E</td>
<td>done</td>
</tr>
<tr>
<td>F</td>
<td>done</td>
</tr>
</tbody>
</table>
Detecting Cycles

Let's simulate BFS in this graph, detecting cycles.
Detecting Cycles

Let's start at E.

Vertex E is visited, enqueued (alone), and is immediately dequeued. Let's call any vertex that has finished those steps “done.”
Detecting Cycles

Vertices \{A, B, D\} are adjacent to E.
Each is visited, its spanning-tree parent (E) is identified, put in the queue.
Detecting Cycles

We dequeue the next vertex, which might be B, and examine its neighbors.

Each *unvisited* neighbor is visited, its parent identified, and enqueued. Already-visited neighbors are connected by *cross edges*.

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>enqueued</td>
</tr>
<tr>
<td>B</td>
<td>done</td>
</tr>
<tr>
<td>C</td>
<td>enqueued</td>
</tr>
<tr>
<td>D</td>
<td>enqueued</td>
</tr>
<tr>
<td>E</td>
<td>done</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
Detecting Cycles

Next, dequeue A and examine its neighbors.

Vertex D is already enqueued so edge \{A,D\} is a cross edge. Vertex F was not visited, so we visit it and enqueue it.
Detecting Cycles

Note that we didn't have to do anything to D – it is already marked as “enqueued”.

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>done</td>
</tr>
<tr>
<td>B</td>
<td>done</td>
</tr>
<tr>
<td>C</td>
<td>enqueued</td>
</tr>
<tr>
<td>D</td>
<td>enqueued</td>
</tr>
<tr>
<td>E</td>
<td>done</td>
</tr>
<tr>
<td>F</td>
<td>enqueued</td>
</tr>
</tbody>
</table>
Next dequeue D and examine its neighbors. This finds another cross edge.
Detecting Cycles

Then dequeue C, with nothing to do there. Same with F.
Detecting Cycles

- Every edge can be identified as a tree edge in the BFS spanning tree, or a cross edge, connecting two siblings or cousins.