Connected Components, Directed Graphs, Topological Sort
How do we tell if two vertices are connected?

A connected to F?
A connected to L?
Connectivity

A graph is *connected* if and only if there exists a path between every pair of distinct vertices.

A graph is connected if and only if there exists a simple path between every pair of distinct vertices

- since every non-simple path contains a cycle, which can be bypassed

**How to check for connectivity?**
- Run BFS or DFS (using an arbitrary vertex as the source)
- If all vertices have been visited, the graph is connected.
- Running time? $O(n + m)$
Connected Components
Subgraphs

A graph $H(V_H, E_H)$ is a subgraph of $G(V_G, E_G)$ if and only if $V_H \subset V_G$ and $E_H \subset E_G$. 
Connected Components

Formal definition

- A connected component is a **maximal connected subgraph** of a graph

- The set of connected components is unique for a given graph

3 components: C1, C2, and C3
Finding Connected Components

Algorithm \textit{DFSC}\textit{onn}(G)
\textbf{Input:} a graph \( G \)
\textbf{Output:} the connected components
1. \textbf{for} each vertex \( v \)
2. \quad \textbf{do} \( \text{flag}[v] := \text{false}; \)
3. \quad \textbf{for} each vertex \( v \)
4. \quad \quad \textbf{do} \text{if} \( \text{flag}[v] = \text{false} \)
5. \quad \quad \text{then} \text{output } "\text{A new connected component:}";
6. \quad \text{\textbf{RDFS}(v)}; \quad \text{Call DFS}

This will find all vertices connected to "\( v \)" \( \Rightarrow \) one connected component

Algorithm \textit{RDFS}(v)
1. \( \text{flag}[v] := \text{true}; \)
2. \text{output } v;
3. \textbf{for} each neighbor \( w \) of \( v \)
4. \quad \textbf{do} \text{if} \( \text{flag}[w] = \text{false} \)
5. \quad \text{then} \text{\textit{RDFS}(w)};

Basic DFS algorithm
Time Complexity

Running time for each $i$ connected component

\[ O(n_i + m_i) \]

Running time for the graph $G$

\[ \sum_i O(n_i + m_i) = O(\sum_i n_i + \sum_i m_i) = O(n + m) \]

Reason: Can two connected components share
- the same edge?
- the same vertex?
Trees

- Tree arises in many computer science applications

- A graph $G$ is a tree if and only if it is connected and acyclic
  (Acyclic means it does not contain any simple cycles)

- The following statements are equivalent
  - $G$ is a tree
  - $G$ is acyclic and has exactly $n-1$ edges
  - $G$ is connected and has exactly $n-1$ edges
Tree Example

- Is it a graph?
- Does it contain cycles? In other words, is it acyclic?
- How many vertices?
- How many edges?
Directed Graph

- A graph is directed if direction is assigned to each edge.
- Directed edges are denoted as *arcs*.
  - Arc is an *ordered pair* \((u, v)\)

Recall: for an undirected graph

- An edge is denoted \(\{u, v\}\), which actually corresponds to two arcs \((u, v)\) and \((v, u)\)
Representations

The adjacency matrix and adjacency list can be used.

1. Adjacency Matrix

2. Adjacency List
Directed Acyclic Graph

- A **directed path** is a sequence of vertices \((v_0, v_1, \ldots, v_k)\)
  - Such that \((v_i, v_{i+1})\) is an *arc*

- A **directed cycle** is a directed path such that the first and last vertices are the same.

- A directed graph is **acyclic** if it does not contain any directed cycles
Indegree and Outdegree

Since the edges are directed

- We can’t simply talk about $\text{Deg}(v)$

Instead, we need to consider the arcs coming “in” and going “out”

- Thus, we define terms $\text{Indegree}(v)$, and $\text{Outdegree}(v)$

Each arc $(u,v)$ contributes count 1 to the outdegree of $u$ and the indegree of $v$

\[
\sum_{\text{vertex } v} \text{indegree (v)} = \sum_{\text{vertex } v} \text{outdegree (v)} = m
\]
Calculate Indegree and Outdegree

**Outdegree** is simple to compute
- Scan through list Adj[v] and count the arcs

**Indegree calculation**
- First, initialize indegree[v]=0 for each vertex v
- Scan through adj[v] list for each v
  - For each vertex w seen, indegree[w]++;
- Running time: $O(n+m)$
Example

Indeg(2)?
Indeg(8)?
Outdeg(0)?
Num of Edges?
Total OutDeg?
Total Indeg?
Directed Graphs Usage

- Directed graphs are often used to represent order-dependent tasks
  - That is we cannot start a task before another task finishes

- We can model this task dependent constraint using arcs
- An arc \((i,j)\) means task \(j\) cannot start until task \(i\) is finished

- Clearly, for the system not to hang, the graph must be acyclic
University Example

CS departments course structure

Any directed cycles?
How many indeg(171)?
How many outdeg(171)?
Topological Sort

- Topological sort is an algorithm for a directed acyclic graph.
- Linearly order the vertices so that the linear order respects the ordering relations implied by the arcs.

For example:

0, 1, 2, 5, 9
0, 4, 5, 9
0, 6, 3, 7 ?

It may not be unique as they are many ‘equal’ elements!
Topological Sort Algorithm

Observations

- Starting point must have zero indegree
- If it doesn’t exist, the graph would not be acyclic

Algorithm

1. A vertex with zero indegree is a task that can start right away. So we can output it first in the linear order.
2. If a vertex $i$ is output, then its outgoing arcs $(i, j)$ are no longer useful, since tasks $j$ does not need to wait for $i$ anymore- so remove all $i$’s outgoing arcs
3. With vertex $i$ removed, the new graph is still a directed acyclic graph. So, repeat step 1-2 until no vertex is left.
Topological Sort

Algorithm $TSort(G)$
Input: a directed acyclic graph $G$
Output: a topological ordering of vertices
1. initialize $Q$ to be an empty queue;
2. for each vertex $v$
   3. do if $\text{indegree}(v) = 0$
   4. then enqueue($Q, v$);
5. while $Q$ is non-empty
6. do $v ::= \text{dequeue}(Q)$;
7. output $v$;
8. for each arc $(v, w)$
9. do $\text{indegree}(w) = \text{indegree}(w) - 1$;
10. if $\text{indegree}(w) = 0$
11. then enqueue($w$)

The running time is $O(n + m)$. 
Example

Q = { 0 }

<table>
<thead>
<tr>
<th>Indegree</th>
<th>start</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>
Deque 0  Q = {} 
-> remove 0’s arcs – adjust indegrees of neighbors

OUTPUT:  0
Q = \{ 6, 1, 4 \}
Enqueue all starting points

OUTPUT: 0
Dequeue 6  Q = { 1, 4 }  
Remove arcs .. Adjust indegrees of neighbors

OUTPUT: 0 6
Q = { 1, 4, 3 }  
Enqueue 3

OUTPUT: 0 6
Dequeue 1  \( Q = \{ 4, 3 \} \)
Adjust indegrees of neighbors

OUTPUT: 0 6 1
Deque 1  Q = { 4, 3, 2 }
Enqueue 2

OUTPUT:  0 6 1
Deque 4  Q = { 3, 2 }
Adjust indegrees of neighbors

OUTPUT: 0 6 1 4
Dequeue 4  Q = {  3, 2 }  
No new start points found

OUTPUT:  0 6 1 4
Dequeue 3  Q = { 2 }
Adjust 3’s neighbors

OUTPUT: 0 6 1 4 3
Deque 3  \( Q = \{ 2 \} \)
No new start points found

OUTPUT: 0 6 1 4 3
Dequeue 2  Q = {}  
Adjust 2’s neighbors  

OUTPUT:  0 6 1 4 3 2
Dequeue 2  Q = { 5, 7 }  
Enqueue 5, 7  

OUTPUT:  0 6 1 4 3 2
Dequeue 5  Q = { 7 }  
Adjust neighbors

OUTPUT: 0 6 1 4 3 2 5
Dequeque 5  Q = { 7 }  
No new starts

OUTPUT: 0 6 1 4 3 2 5
Dequeue 7  Q = { }  
Adjust neighbors

OUTPUT: 0 6 1 4 3 2 5 7
Dequeue 7  Q = { 8 }  
Enqueue 8  

OUTPUT: 0 6 1 4 3 2 5 7
Deque 8  Q = {}  
Adjust indegrees of neighbors

OUTPUT: 0 6 1 4 3 2 5 7 8
Dequeue 8  \( Q = \{ 9 \} \)
Enqueue 9
Dequeue 9  \( Q = \{ \} \)
STOP – no neighbors

OUTPUT: 0 6 1 4 3 2 5 7 8 9
OUTPUT: 0 6 1 4 3 2 5 7 8 9

Is output topologically correct?
Topological Sort: Complexity

- We never visited a vertex more than one time.

- For each vertex, we had to examine all outgoing edges.
  - $\Sigma \text{outdegree}(v) = m$
  - This is summed over all vertices, not per vertex.

- So, our running time is exactly $O(n + m)$. 
Summary: Two representations:

- **Some definitions:** ...

- **Two sizes:** \( n = |V| \) and \( m = |E| \),
  - \( m = O(n^2) \)

- **Adjacency List**
  - More compact than adjacency matrices if graph has few edges
  - Requires a scan of adjacency list to check if an edge exists
  - Requires a scan to obtain all edges!

- **Adjacency Matrix**
  - Always require \( n^2 \) space
    - This can waste a lot of space if the number of edges are sparse
  - find if an edge exists in \( O(1) \)
  - Obtain all edges in \( O(n) \)

- \( O(n+m) \) for indegree for a DAG
(one), Two, (three) algorithms:

BFS (queue)

s is visited
enqueue(Q,s)
while not-empty(Q)
    v <- dequeue(Q)
    W = \{unvisited neighbors of v\}
    for each w in W
        w is visited
        enqueue(Q,w)

DFS (stack)

s is visited
push(S,s)
while not-empty(S)
    v <- pop(S)
    W = \{unvisited neighbors of v\}
    for each w in W
        w is visited
        push(S,w)

RDFS(v)

v is visited
W = \{unvisited neighbors of v\}
for each w in W
    RDFS(w)
Two applications

☞ For each non-visited vertex, run ‘connected component’ (either BFS or DFS)
  ■ For a connected component, list all vertices, find a spanning tree (BFS tree or DFS tree)
☞ ‘Shortest paths’ and ‘topological sort’ (for DAG only) are close to BFS