Last Time

• Graphs
• Explore / DFS
• Graph Runtimes
explore(v)
    v.visited ← true
    For each edge (v, w)
        If not w.visited
            explore(w)

DepthFirstSearch(G)
    Mark all v ∈ G as unvisited
    For v ∈ G
        If not v.visited, explore(v)

O(|V| + |E|)
Graph algorithm runtimes depend on both $|V|$ and $|E|$. (Note $O(|V|+|E|)$ is linear time)

What algorithm is better may depend on relative sizes of these parameters.

**Sparse Graphs:**
$|E|$ small ($\approx V$)
Examples:
- Internet
- Road maps

**Dense Graphs:**
$|E|$ large ($\approx V^2$)
Examples:
- Flight maps
- Wireless networks
Question: Graph Runtimes

Suppose that you have two graph algorithms for the same problem
Alg1 has runtime $O(|V|^{3/2})$
Alg2 has runtime $O(|E|)$
Which of the following is likely true?
A) Alg1 is faster on most graphs
B) Alg2 is faster on most graphs
C) Alg1 is faster on sparse graphs, slower on dense graphs
D) Alg2 is faster on sparse graphs, slower on dense graphs
Today

• Graph Representations
• Connected Components
• Pre- & Post-orders
Graph Representations

How do you store a graph in a computer?

- **Adjacency matrix**: Store list of vertices and an array $A[i,j] = 1$ if edge between $v_i$ and $v_j$.
  - Small space for dense graphs.
  - Slow for most operations.

- **Edge list**: List of all vertices, list of all edges
  - Hard to determine edges out of single vertex.

- **Adjacency list**: For each vertex store list of neighbors.
  - Needed for DFS to be efficient
  - We will usually assume this representation
Connected Components

• Want to understand which vertices are reachable from which others in graph G.

• \texttt{explore}(v) finds which vertices are reachable from a given vertex.

\textbf{Theorem:} The vertices of a graph G can be partitioned into \textit{connected components} so that \(v\) is reachable from \(w\) if and only if they are in the same connected component.
Example
Question: Connected Components

How many connected components does the graph below have?

A) 0
B) 1
C) 2
D) 3
E) 4
Problem: Computing Connected Components

Given a graph G, compute its connected components.

**Easy:** For each v, run \texttt{explore(v)} to find vertices reachable from it. Group together into components.

Runtime: \( O(|V|(|V|+|E|)) \).

**Better:** Run \texttt{explore(v)} to find the component of v. Repeat on unclassified vertices.
DFS lets us do this!

ConnectedComponents(G)

CCNum ← 0
For v ∈ G
    v.visited ← false
For v ∈ G
    If not v.visited
        CCNum++
        explore(v)

explore(v)
    v.visited ← true
    v.CC ← CCNum
    For each edge (v, w)
        If not w.visited
            explore(w)

Runtime O(|V|+|E|).
Example

CCNum: 4

1 1 1 1
1
1
1
3 3 4

A
B
C
H

D
E
F
G

1 1 1
2 2 2
2 2 2
2
Discussion about DFS

What does DFS actually do?

• No output.
• Marks all vertices as visited.
• Easier ways to do this.

However, DFS also is a useful way to explore the graph. By augmenting the algorithm a bit (like we did with the connected components algorithm), we can learn useful things.
Pre- and Post- Orders

Augment how?

• Keep track of what algorithm does & in what order.
• Have a “clock” and note time whenever:
  – Algorithm visits a new vertex for the first time.
  – Algorithm finishes processing a vertex.
• Record values as \( v.pre \) and \( v.post \).
Computing Pre- & Post- Orders

**DFS(G)**

\[
\text{clock} \leftarrow 1 \\
\text{For } v \in G \\
\quad v.\text{visited} \leftarrow \text{false} \\
\text{For } v \in G \\
\quad \text{If not } v.\text{visited} \\
\quad \quad \text{explore}(v) \\
\]

**explore(v)**

\[
\quad v.\text{visited} \leftarrow \text{true} \\
\quad v.\text{pre} \leftarrow \text{clock} \\
\quad \text{clock}++ \\
\text{For each edge } (v,w) \\
\quad \quad \text{If not } w.\text{visited} \\
\quad \quad \quad \text{explore}(w) \\
\quad v.\text{post} \leftarrow \text{clock} \\
\quad \text{clock}++
\]

Runtime $O(|V| + |E|)$. 
Example
What do these orders tell us?

**Prop:** For vertices $v$, $w$ consider intervals $[v.\text{pre}, v.\text{post}]$ and $[w.\text{pre}, w.\text{post}]$. These intervals:

1. Contain each other if $v$ is an ancestor/descendant of $w$ in the DFS tree.
2. Are disjoint if $v$ and $w$ are cousins in the DFS tree.
3. Never interleave $(v.\text{pre} < w.\text{pre} < v.\text{post} < w.\text{post})$
Proof

• Assume algorithm finds v before w
  \[(v.pre < w.pre)\]

• If algorithm discovers w after fully processing v:
  – \(v.post < w.pre\)
  – Intervals disjoint
  – v and w are cousins

• If algorithm discovers w before fully processing v:
  – Algorithm finishes processing w before it finishes v
  – \(v.pre < w.pre < w.post < v.post\)
  – Nested intervals
  – v is ancestor of w
Question: Possible Intervals

Which pairs of pre-post intervals are possible for DFS?

A) [1,2] & [3,4]
B) [1,3] & [2,4]
C) [1,4] & [2,3]
D) [1,5] & [2,4]
E) [1,6] & [2,5]
Directed Graphs

Often an edge makes sense both ways, but sometimes streets are one directional.

**Definition:** A directed graph is a graph where each edge has a direction. Goes *from* $v$ *to* $w$. Draw edges with arrows to denote direction.
Question: Directed Graphs

Which of the following make the most sense as directed rather than undirected graphs:

A) The Internet (links connecting webpages)
B) The Internet (wires connecting servers)
C) Facebook (friendships connecting people)
D) Twitter (followings connecting people)
E) Maps (roads connecting intersections)
DFS on Directed Graphs

- Same code
- Only follow *directed* edges from \( v \) to \( w \).
- Runtime still \( O(|V|+|E|) \)
- \( \text{explore}(v) \) discovers all vertices reachable from \( v \) following only directed edges.
Example