Program Structure and Algorithms

Sid Nath

Lecture 5

Agenda

- Administrative
  - HW2 questions?
  - Quiz 3 today on D/Q
- Lecture
- Quiz

Graph Applications

- Numerous
- Networks: social, transportation, circuits, …

- Internet

- Maps

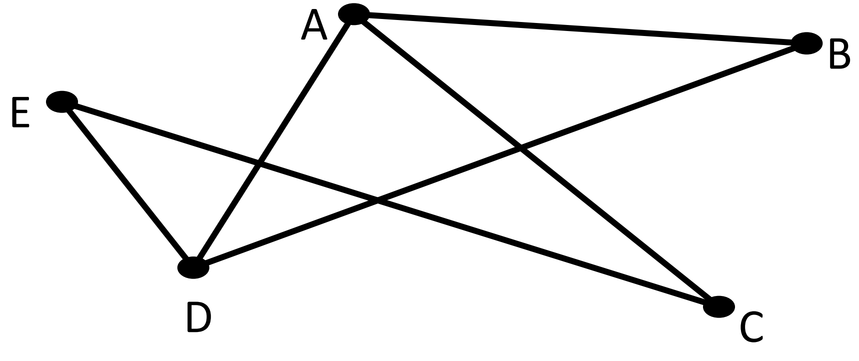
- OS

- Backprop in neural networks

- ….

Graph Definition

- A *graph* G = (V,E) consists of two things:
  - A collection V of *vertices*, or objects to be connected.
  - A collection E of *edges*, each of which connects a pair of vertices.
    - May be undirected or directed
- Examples
  - The internet V = {websites}, E = {links}or V = {computers}, E = {physical connections}
  - Highway system, V = {intersections}, E = {roads}



V = {A,B,C,D,E}
E = {AB, AC, AD, BD, CE, DE}

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

How to Represent Graphs

- Adjacency matrix

```
 0
/ \
1---2
\ /
 3
```

```
# Create a graph with 4 vertices and 5 edges
graph = [[0, 1, 1, 0],
         [1, 0, 1, 1],
         [1, 1, 0, 1],
         [0, 1, 1, 0]]
```

- Adjacency list

```
# Create a graph with 4 vertices and 5 edges
graph = {0: [1, 2],
         1: [0, 2, 3],
         2: [0, 1, 3],
         3: [1, 2]}
```

- Edge list

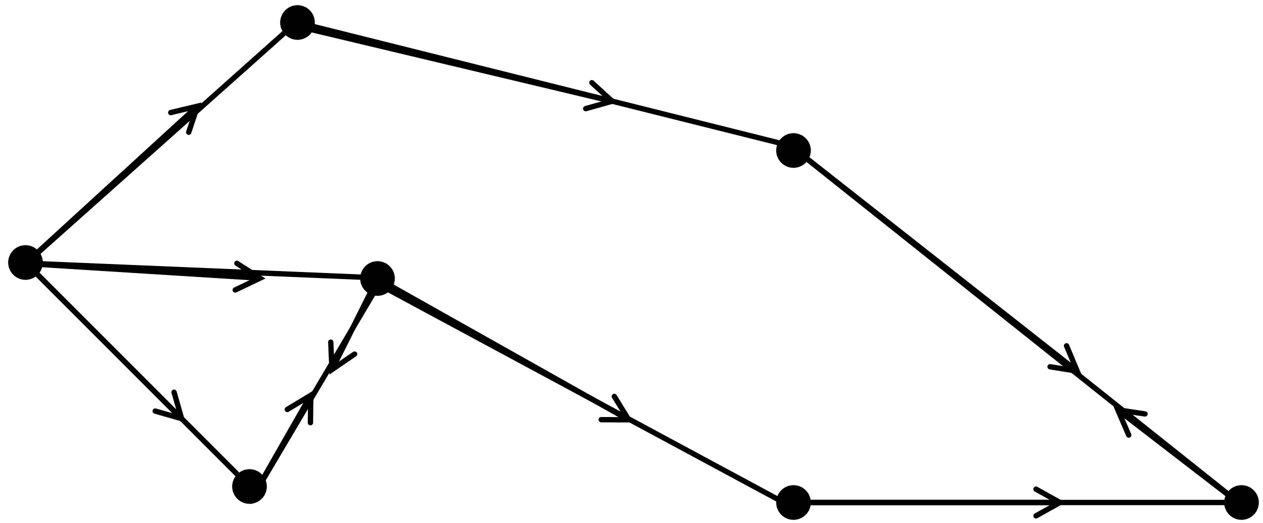
Graph Operations

- Search
  - Vertex reachability
    - Enter/exit times
    - Component the vertex belongs to

Basic Algorithm

Keep track of all areas discovered

While there is an unexplored path,
  follow path



Systematize

Need to keep track of:

- Which vertices discovered
- Which edges have yet to be explored

Explore Algorithm will:

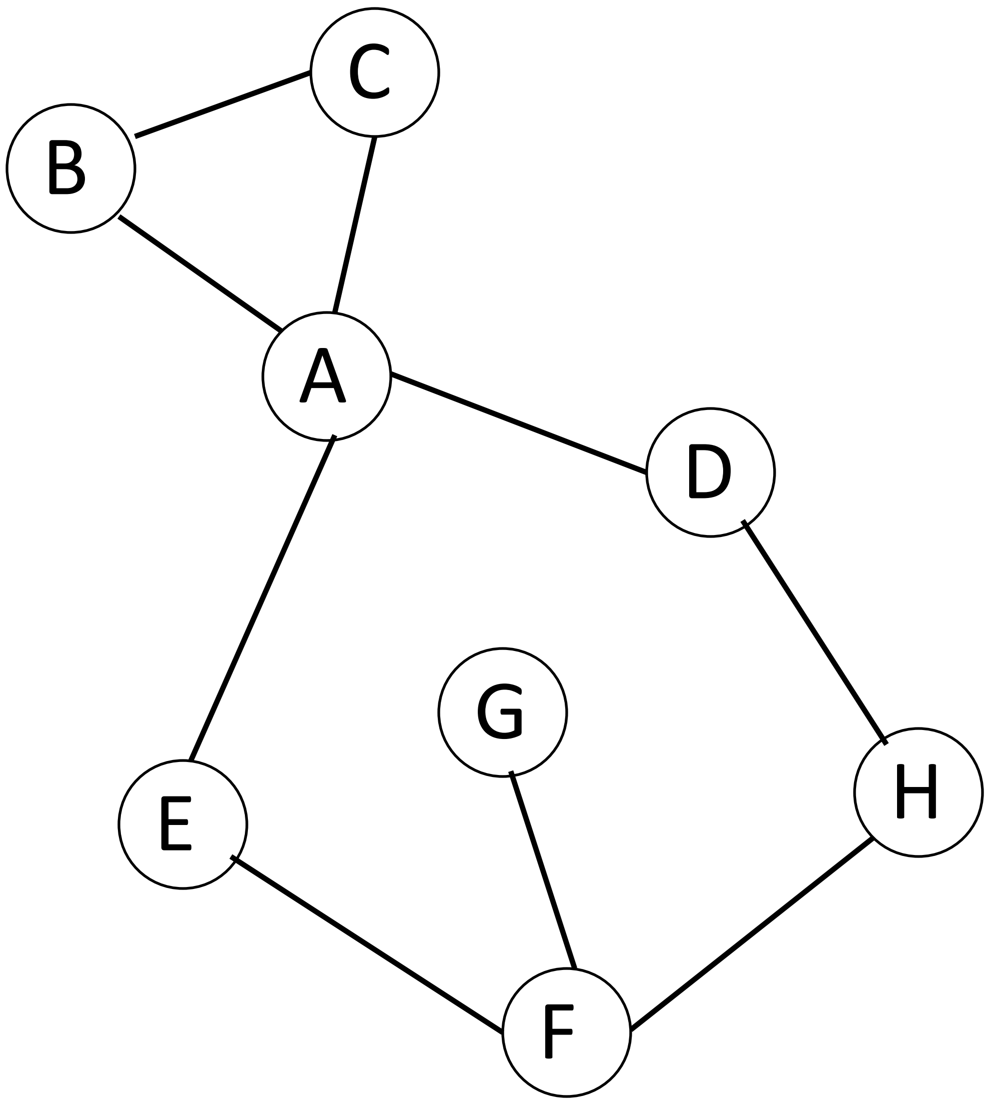
- Use a field `v.visited` to let us know which vertices we have seen.
- Store edges to be explored implicitly in the program stack.

Explore

```
explore(v)
  v.visited ← true
  For each edge (v,w)
    If not w.visited
      explore(w)
      w.prev ← v
```

Example

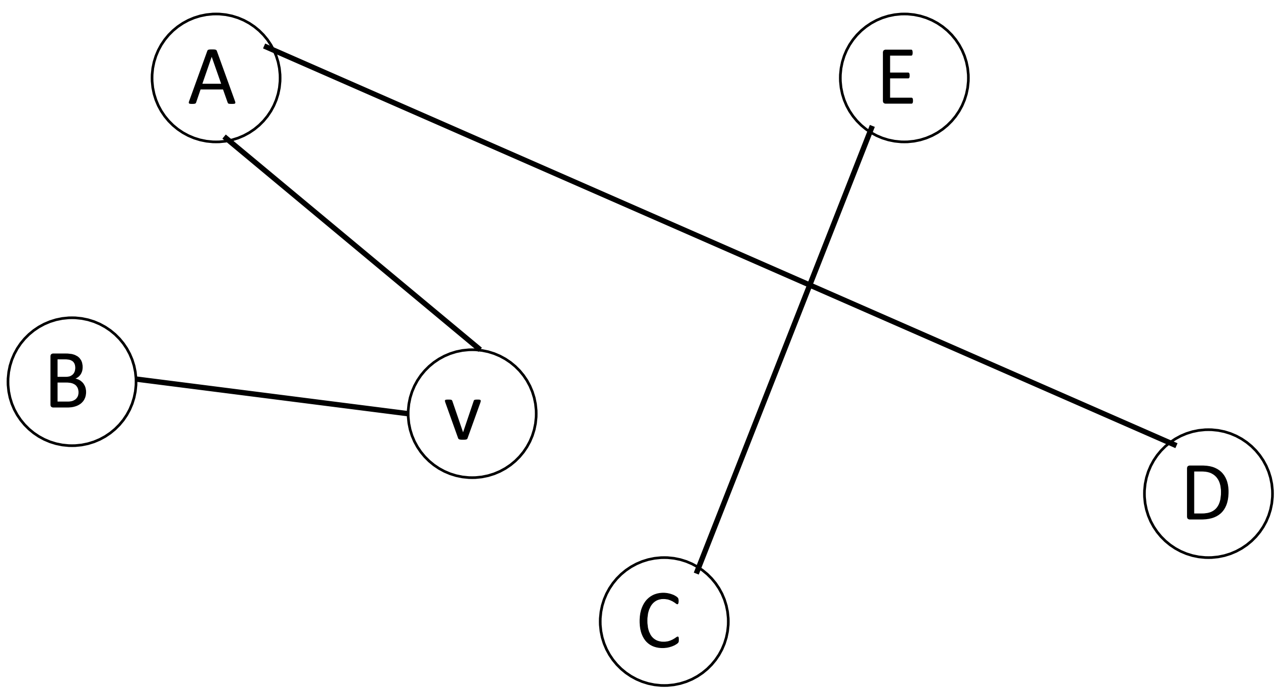
Note: edges used leave behind “DFS tree”.




```
explore(A)
  explore(B)
    explore(A)
    explore(C)
      explore(A)
      explore(B)
  explore(C)
  explore(D)
    explore(A)
    explore(H)
      explore(D)
      explore(F)
        explore(E)
          explore(A)
          explore(F)
        explore(G)
          explore(F)
        explore(H)
  explore(E)
```

Question: explore

Which vertices does `explore(v)` mark as visited?

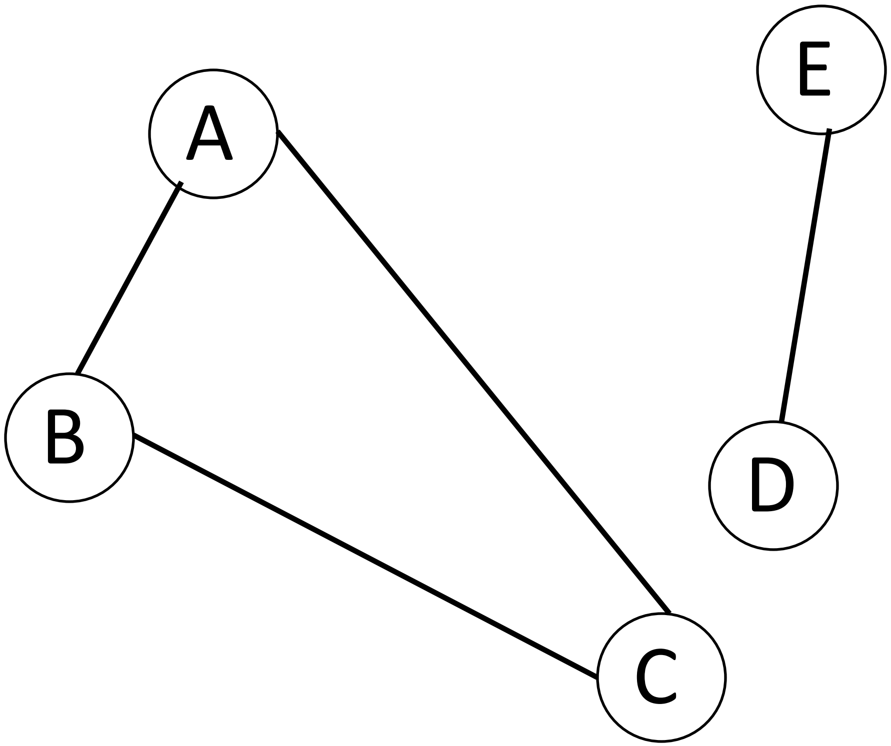


Depth First Search

`explore` only finds the part of the graph reachable from a single vertex. If you want to discover the entire graph, you may need to run it multiple times.

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

Example




```
explore(A)
explore(D)
```

DFS(G) eventually discovers all vertices in G.

Runtime of DFS

```
explore(v)
  v.visited ← true
  For each edge (v,w)
     If not w.visited
          explore(w)
DFS(G)
  Mark all v ∈ G as unvisited
  For v ∈ G
    If not v.visited, explore(v)
```

Run once per vertex — O(|V|) total

Run once per neighboring vertex — O(|E|) total

O(|V|)

Final runtime: O(|V|+|E|)

Note on Graph Algorithm Runtimes

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 (≈ V)

Examples:

- Internet
- Road maps

Dense Graphs:

|E| large (≈ V^2)

Examples:

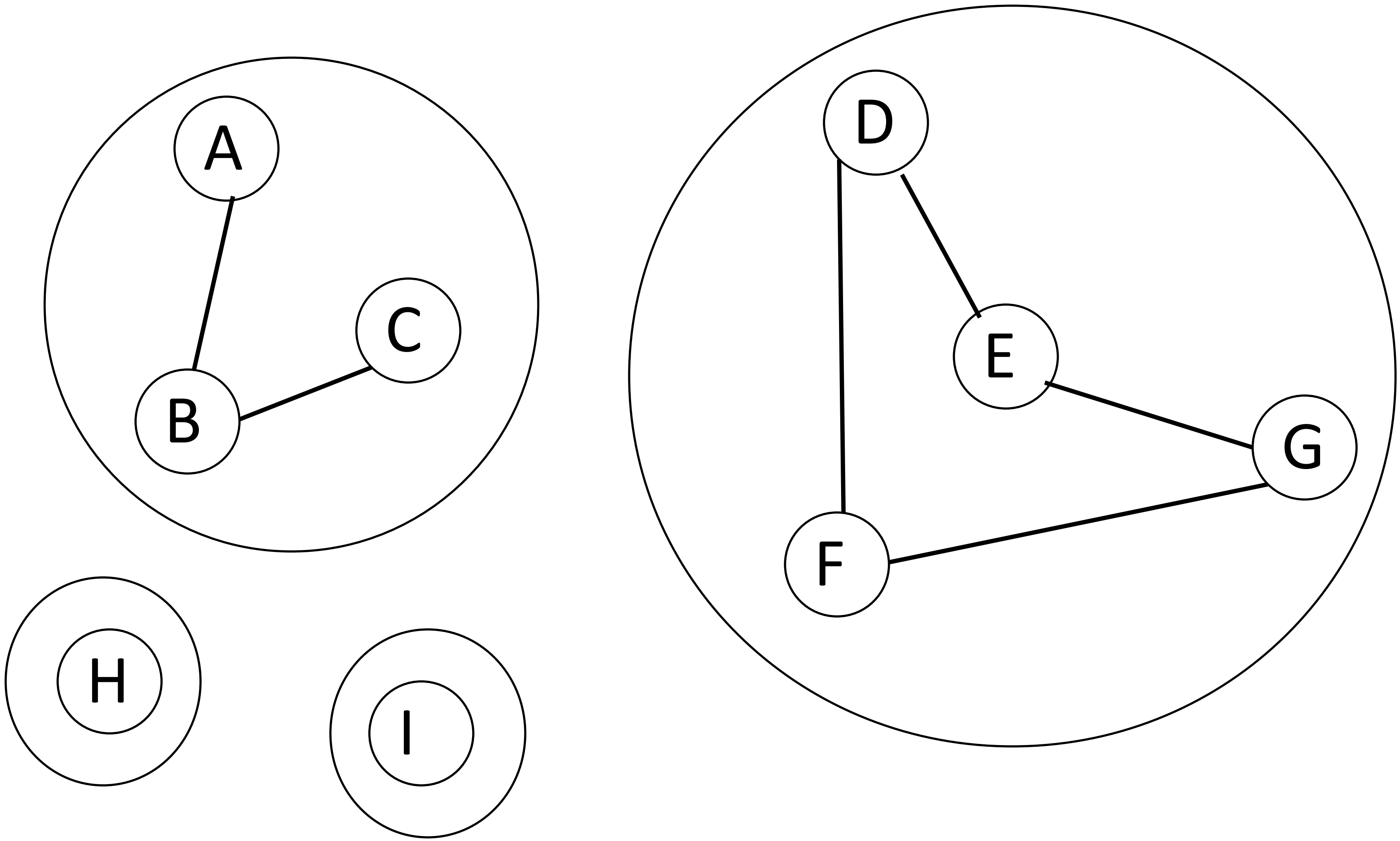
- Flight maps
- Wireless networks

Connected Components

- Want to understand which vertices are reachable from which others in graph G.
- `explore(v)` finds which vertices are reachable from a given vertex.

Theorem: The vertices of a graph G can be partitioned into *connected components* so that v is reachable from w if and only if they are in the same connected component.

Example



Problem: Computing Connected Components

Given a graph G, compute its connected components.

Run `explore(v)` to find the component of v. Repeat on unclassified vertices.

DFS lets us do this!

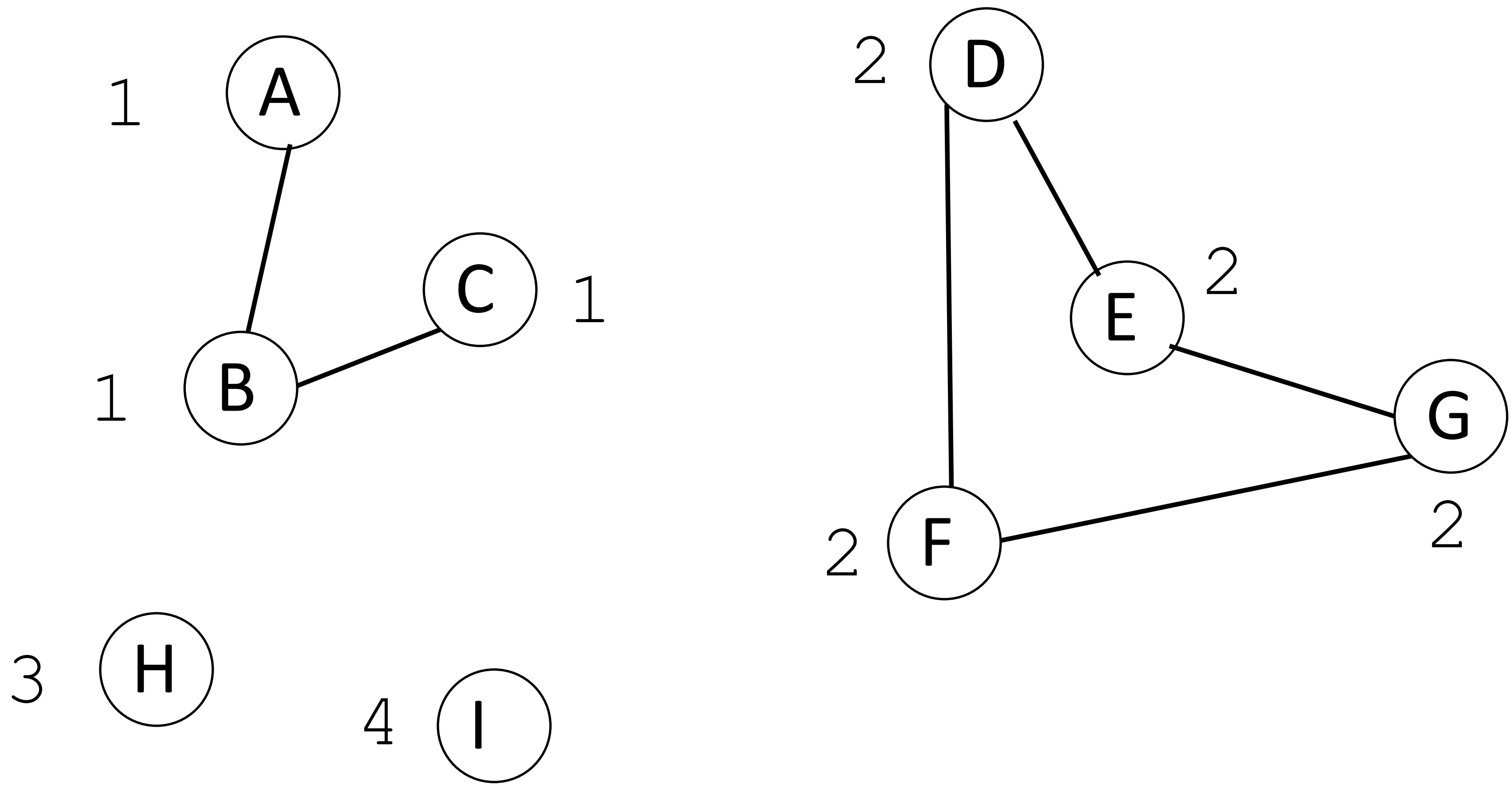
```
ConnectedComponents(G)
  CCNum ← 0
  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      CCNum++
      explore(v)
```

```
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



Discussion about DFS

What does DFS actually *do*?

- No output.
- Marks all vertices as visited.

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

- 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`.

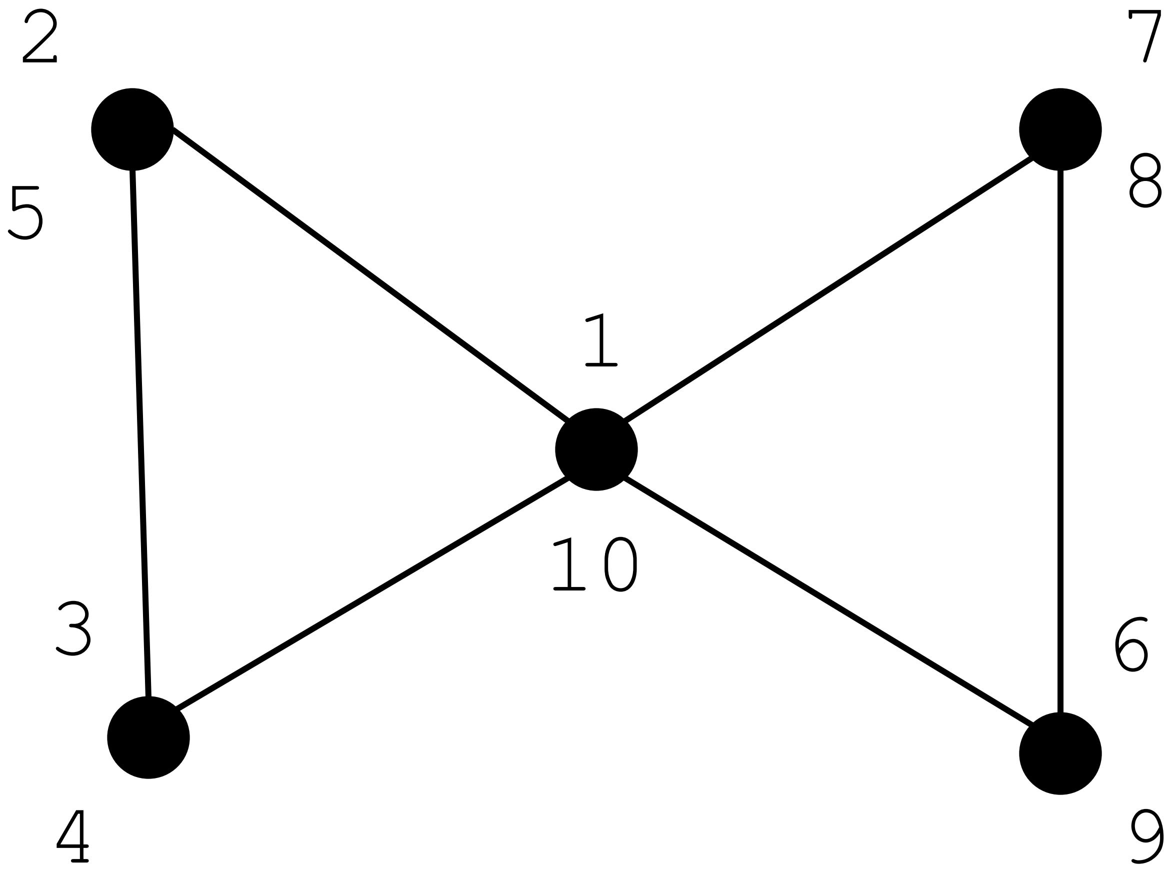
Computing Pre- & Post- Orders

```
DFS(G)
  clock ← 1
  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      explore(v)
```

```
explore(v)
  v.visited ← true
  v.pre ← clock
  clock++
  For each edge (v,w)
    If not w.visited
      explore(w)
  v.post ← clock
  clock++
```

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

Example



What do these orders tell us?

Prop: For vertices v, w consider intervals [`v.pre`, `v.post`] and [`w.pre`, `w.post`].
These intervals:

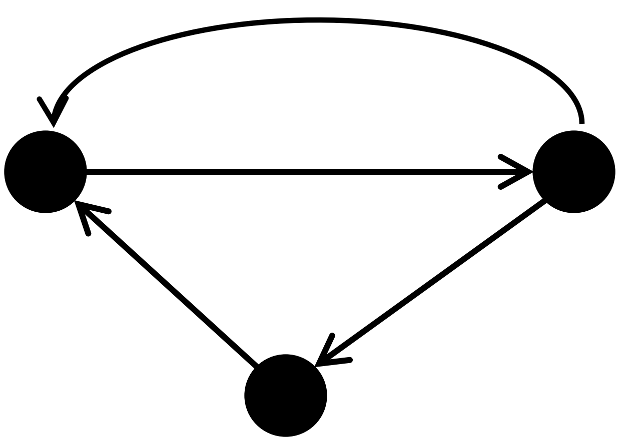
1. Contain each other if v is an ancestor/descendant of w in the DFS tree.
2. Are disjoint if v and w are cousins in the DFS tree.
3. Never interleave
(`v.pre` < `w.pre` < `v.post` < `w.post`)

Directed Graphs

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

<u>Definition:</u> A <u>directed graph</u> is a graph where each edge has a direction. Goes *from* v *to* w.

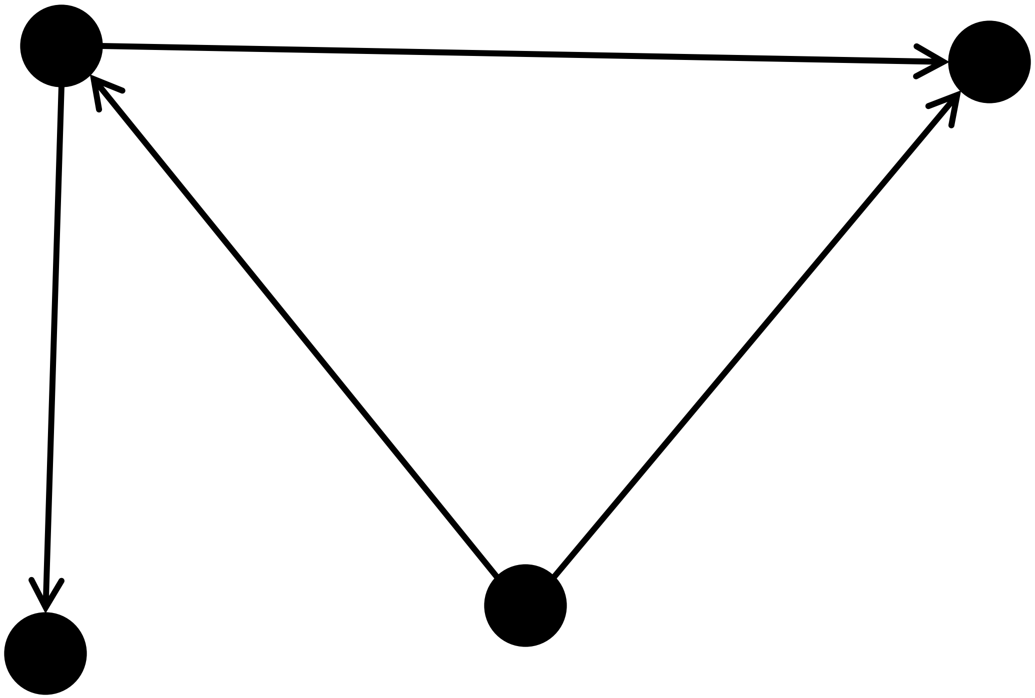
Draw edges with arrows to denote direction.



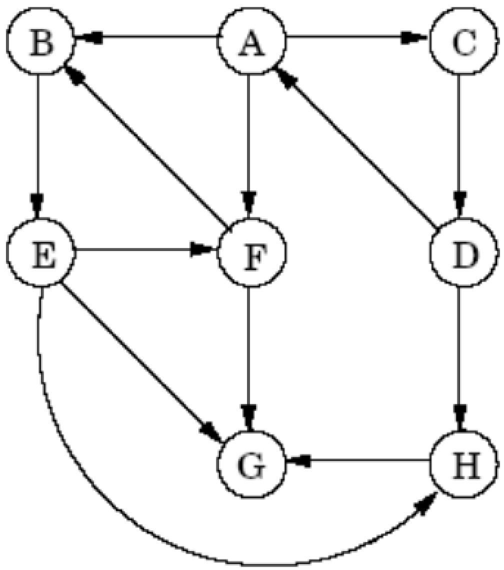
DFS on Directed Graphs

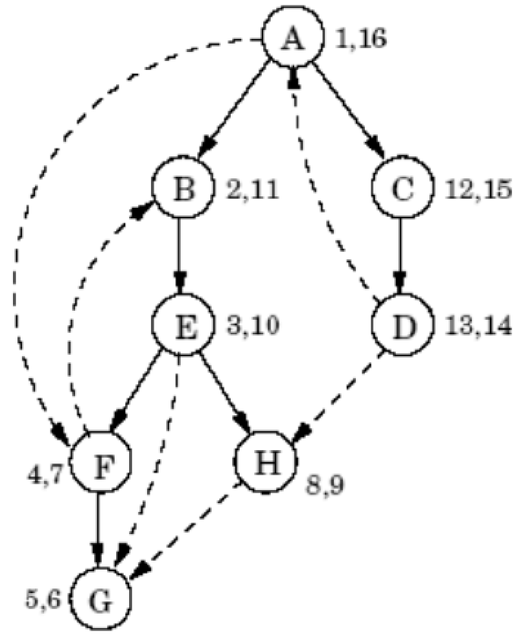
- Same code
- Only follow *directed* edges from v to w.
- Runtime still O(|V|+|E|)
- `explore(v)` discovers all vertices reachable from v following only directed edges.

Example



DFS on Directed Graphs

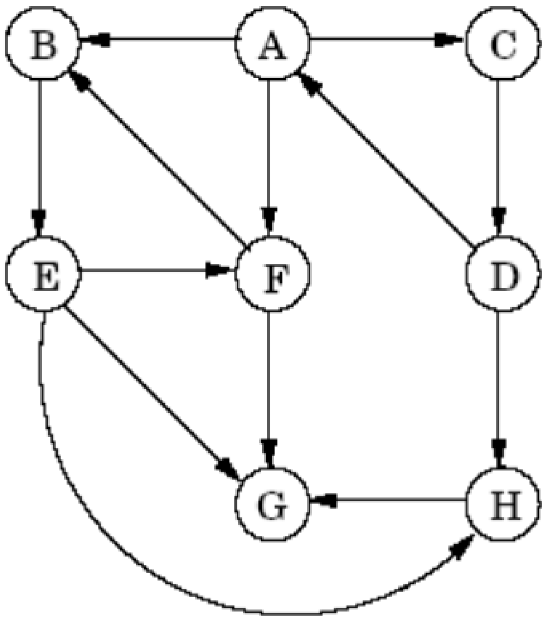


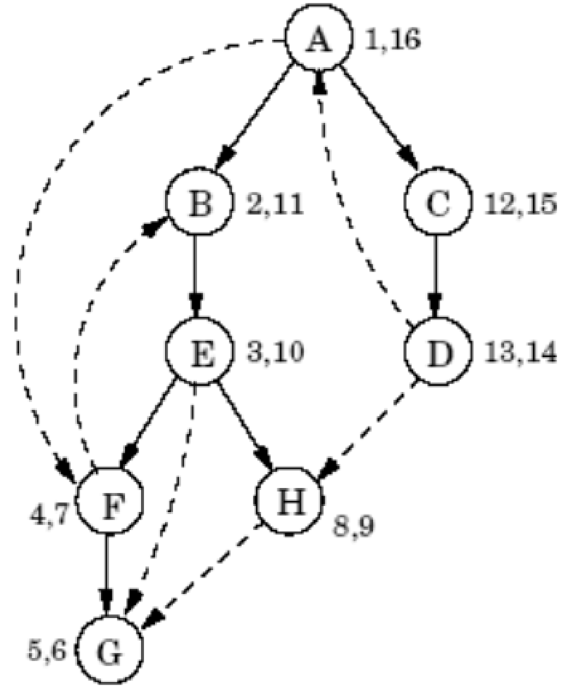


- A is *root* of tree; all other nodes are A's *descendants*
- E has *descendants* F, G, H (E is an *ancestor* of G)
- C is the *parent* of D
- H is a *child* of E

TERMINOLOGY

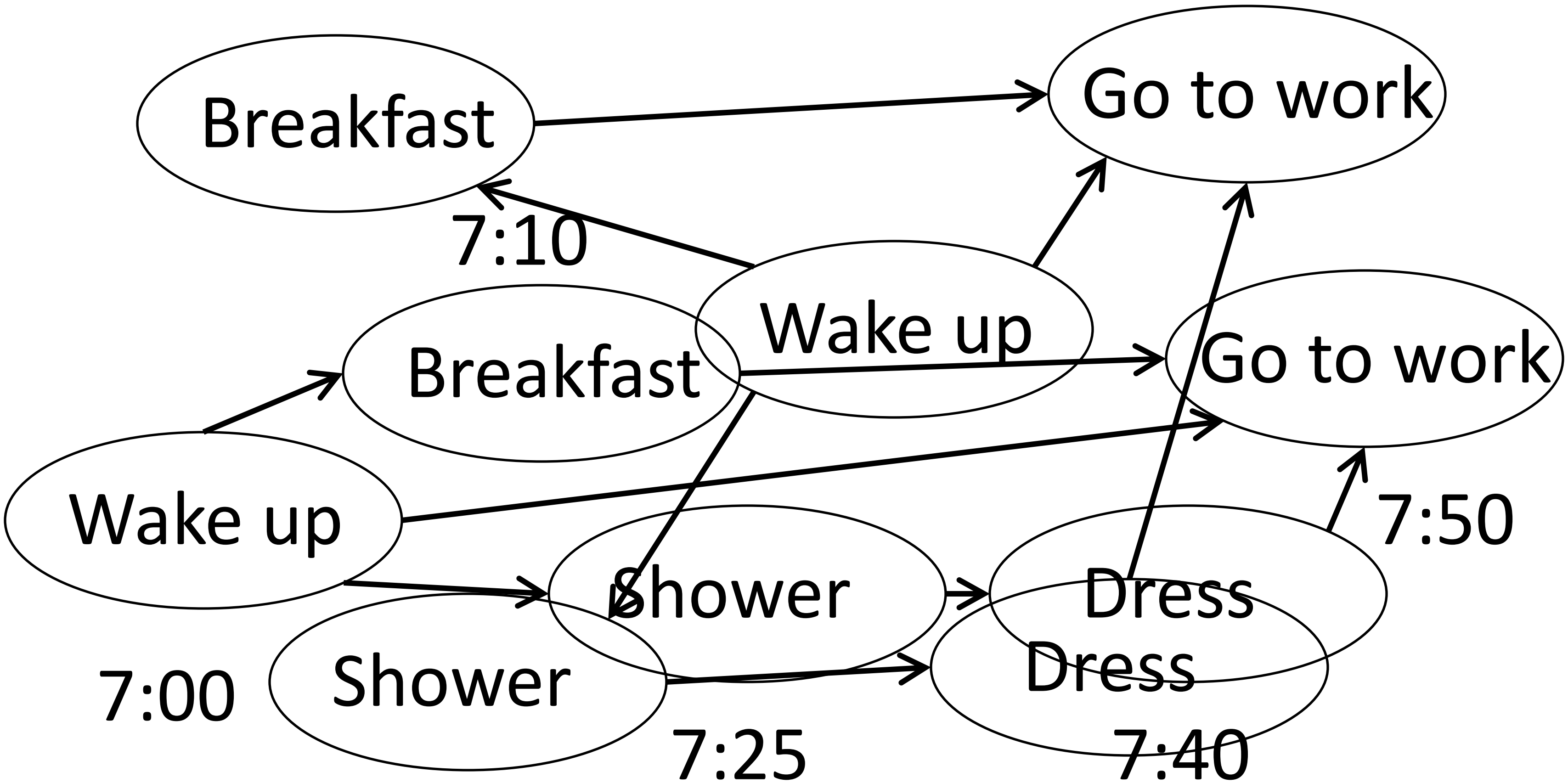
Ancestry and Pre/Post Numbers





- **Node u is an ancestor of node v <u>iff</u> pre(u) < pre(v) < post(u)**
- *Because u is an ancestor of v iff u is discovered first, and then v is encountered during the exploration of u*
- [Def. Node v is a *descendant* of u <u>iff</u> node u is an *ancestor* of v]

Dependency Graphs



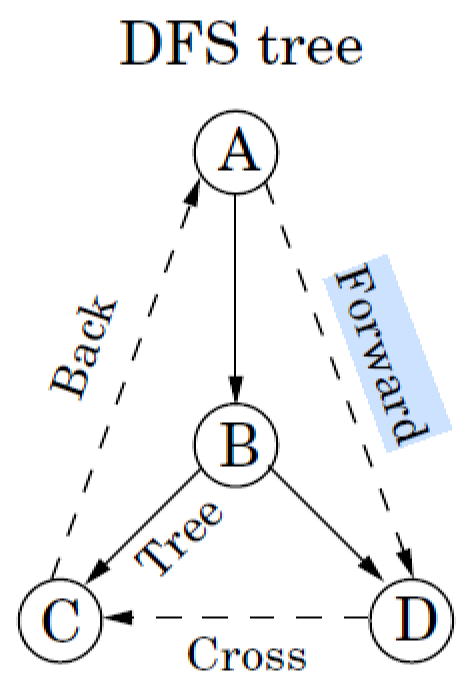
Dependency Graphs

A directed graph can be thought of as a graph of dependencies. Where an edge $v \to w$ means that v should come before w.

Definition: A topological ordering of a directed graph is an ordering of the vertices so that for each edge (v,w), v comes before w in the ordering.

Edge classification in Directed Graphs

- Forward edges are between a node and its non-child descendent
- Back edges lead to an ancestor node
- Cross edges lead to neither ancestor or descendent; they lead to a node that has already been completely explored (i.e., post visited)
- Tree edges are between parent and children.



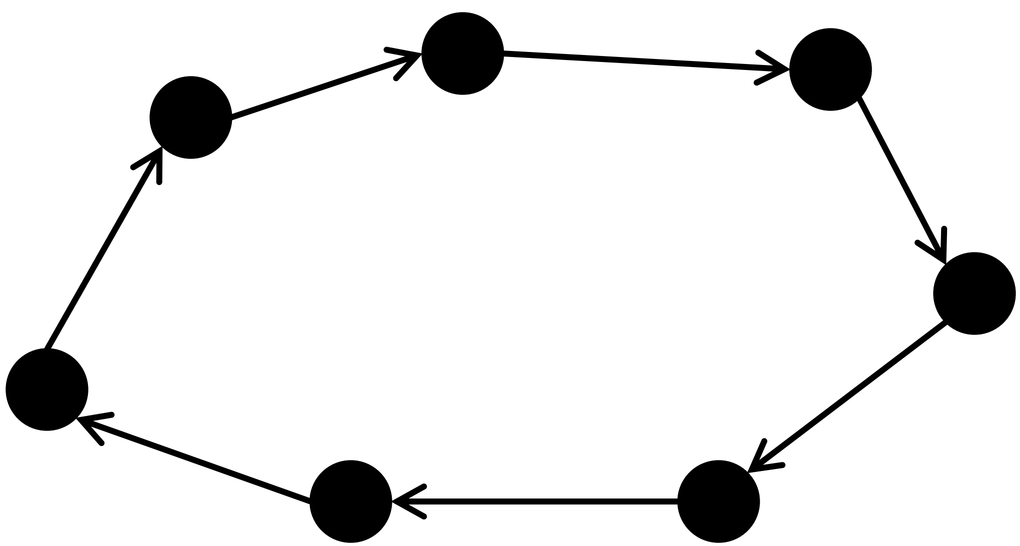
pre/post *ordering for* (u, v)	*Edge type*
$[_u \quad [_v \quad]_v \quad]_u$	Tree/forward
$[_v \quad [_u \quad]_u \quad]_v$	Back
$[_v \quad]_v \quad [_u \quad]_u$	Cross

Cycles

<u>Definition:</u> A <u>cycle</u> in a directed graph is a sequence of vertices

$v_1, v_2, v_3, \ldots, v_n$ so that there are edges

$(v_1,v_2), (v_2,v_3), \ldots, (v_{n-1},v_n), (v_n,v_1)$



Cycles in Directed Graphs

- ***A directed graph has a cycle iff DFS reveals a back edge***
- ($\leftarrow$) If (u,v) is a back edge, then it along with the $v \rightarrow u$ path in the search tree will form a cycle.
- ($\rightarrow$) If the graph has a cycle $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$ then consider the node with smallest *pre* number, call it v_i. All other v_j on the cycle are reachable from v_i and will therefore be descendants of v_i in the search tree. Thus, the edge $v_{i-1} \rightarrow v_i$ is a back edge.

- So, we can determine whether G is acyclic in linear time

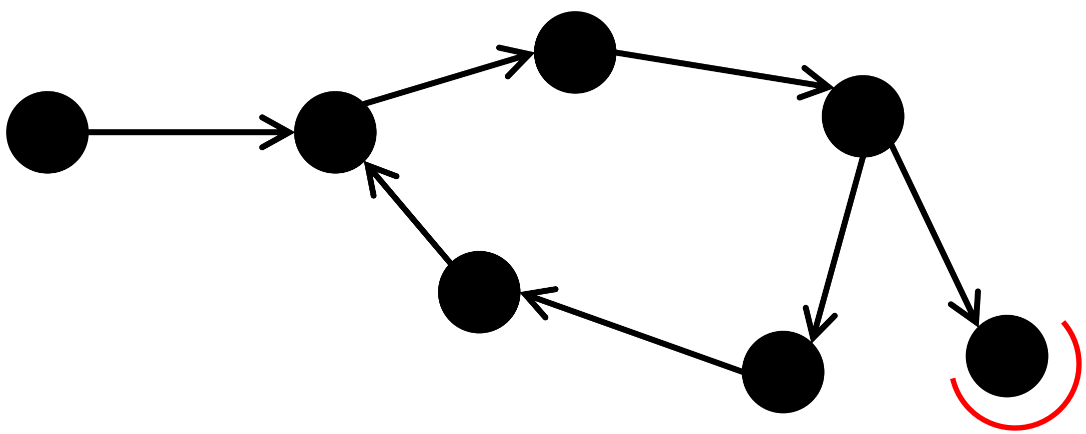
DAGs

<u>Definition:</u> A <u>Directed Acyclic Graph</u> (DAG) is a directed graph which contains no cycles.

Facts:

Let G be a (finite) DAG. Then G has a topological ordering.

Every finite DAG contains at least one sink.



Sources and Sinks in a DAG

- Node with the smallest post number has no outgoing edges.
  - Outdegree = 0
  - Node is called a sink

- Node with the largest post number has no incoming edges
  - Indegree = 0
  - Node is called a source

Topological Ordering and DAGs

- Prelude to shortest paths
- Generic scheduling problem
- Input:
  - Set of tasks $\{T_1, T_2, \ldots, T_n\}$
    - Example: getting dressed in the morning: put on shirt, socks, shoes, ..
  - Set of dependencies $\{T_1 \rightarrow T_2, T_3 \rightarrow T_4, T_5 \rightarrow T_1, \ldots\}$
    - Example: must put on socks before shoes

Topological Ordering and DAGs

- Want
  - Ordering of tasks which is consistent with dependencies
- Problem representation: directed acyclic graph (DAG)
  - Vertices = tasks; directed edges = dependencies
  - Acyclic: if there exists a cycle of dependencies, no solution possible
- General model for causality, dependency

Topological Ordering Algorithm

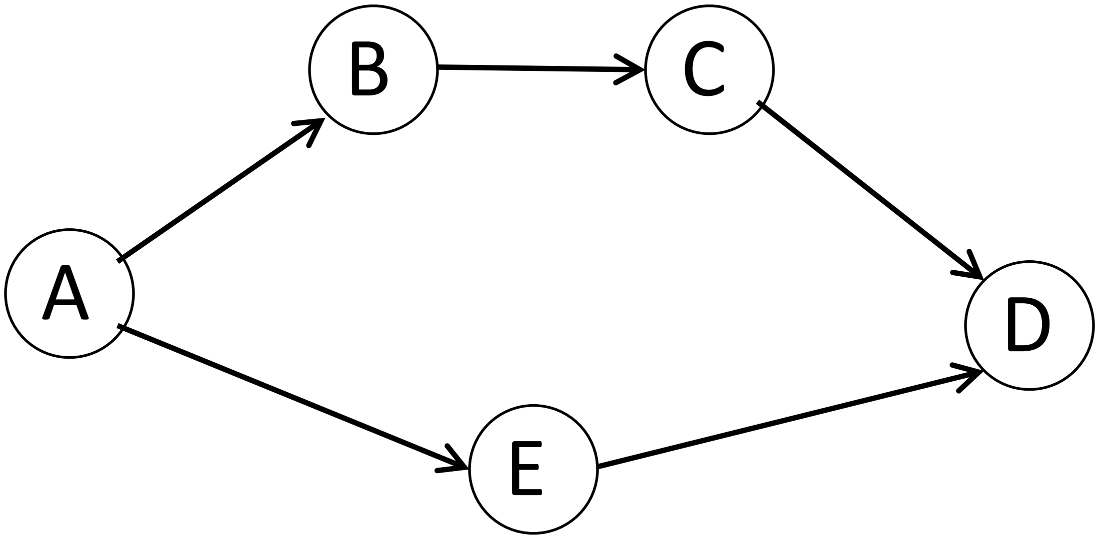
```
TopologicalSort(G)
   Run DFS(G) w/ pre/post numbers
   Return the vertices in reverse
  postorder
```

Note: Can add vertices to list as postorder assigned.

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

Topological Ordering

Problem: Design an algorithm that given a DAG G computes a topological ordering on G.



This is just DFS ordering!

Final Ordering: A E B C D

Topological Sort of a DAG

Useful algorithm.

- Linear ordering of vertices s.t. v → w, implies that v appears before w
- In a DAG, every edge leads to a vertex with lower post number. Why?
  - Any edge (u, v) for which post(v) > post(u) is a back edge.
  - Does a DAG have back edges?
- Many graph algorithms are relatively easy to find the answer for v if you’ve already found the answer for everything downstream of v.
  - Topologically sort G.
  - Solve for v in reverse topological order.

Lecture 5 summary

- Graphs
- Depth First Search
- DAGs and topo sorting