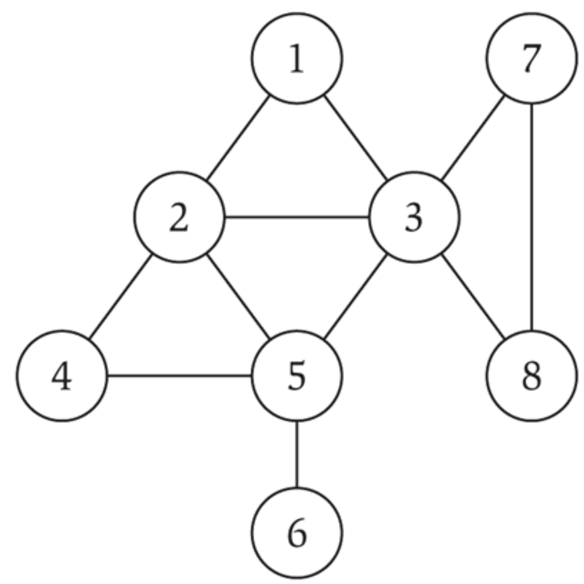
CS 256
Graph Traversals

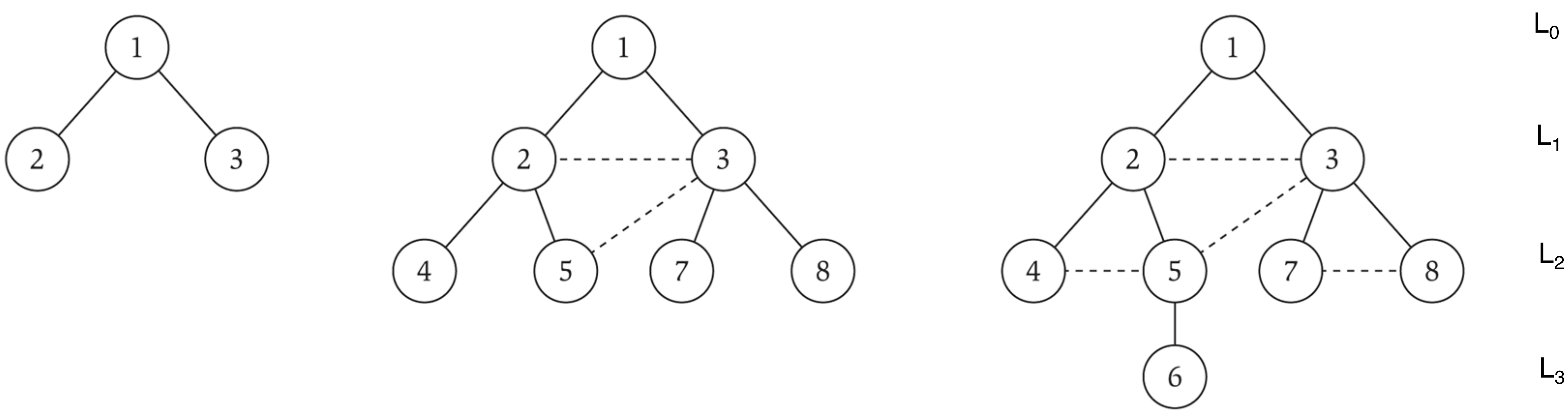
Admin

- Assignment 1 is out
  - Start soon!
- Finish up Assignment 0
  - Slack
- In-person class hopefully starts Monday
  - I’ll send an email over the weekend
- Colloquium 3:15 PM today: what other students did in industry over the summer

BFS Tree Structure

- Property. Let T be a BFS tree of $G = (V, E)$, and let (x, y) be an edge of G. Then, the levels of x and y differ by at most 1.

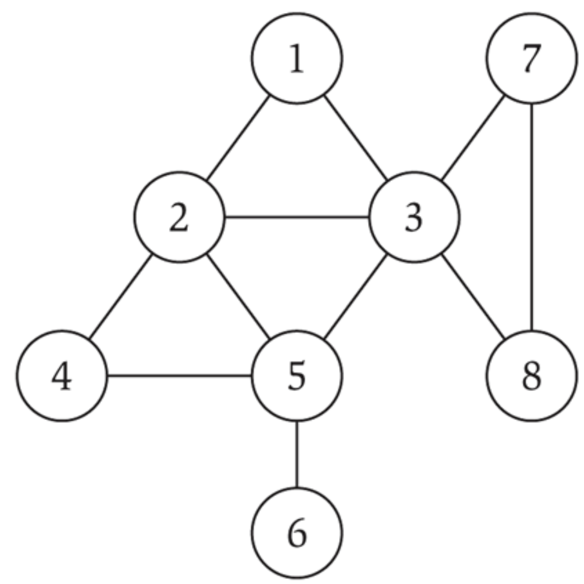


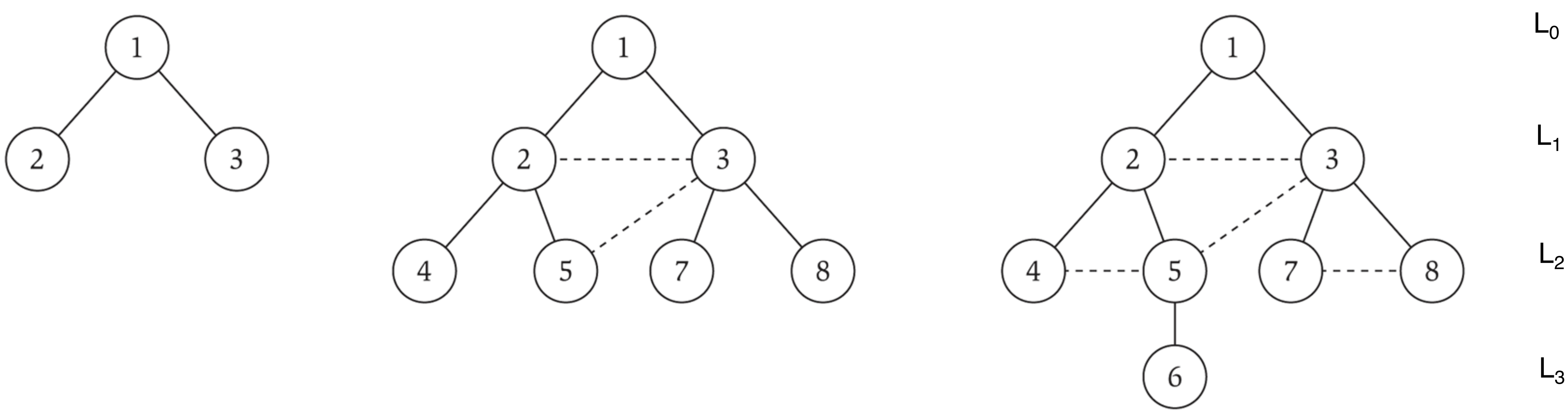


(a) (b) (c)

BFS Tree Structure

- Property. Let T be a BFS tree rooted at r of a connected unweighted graph, then the path from r to any node $u \in V$ in T is **the shortest path** from r to u.





(a) **(b)** **(c)**

Spanning Trees

- **Definition.** A **spanning tree** of an undirected graph G is a connected acyclic subgraph of G that contains every node of G.
- The tree produced by the BFS algorithm (with $((u, \text{parent}(u))$ as edges) is a spanning tree of the component containing s.
- **Connected component of s:** all nodes reachable from s
- In an undirected graph, a BFS spanning tree gives the shortest path from s to every other vertex in its component
- (We will revisit shortest path in a couple of lectures)
- BFS trees in general are short and thick

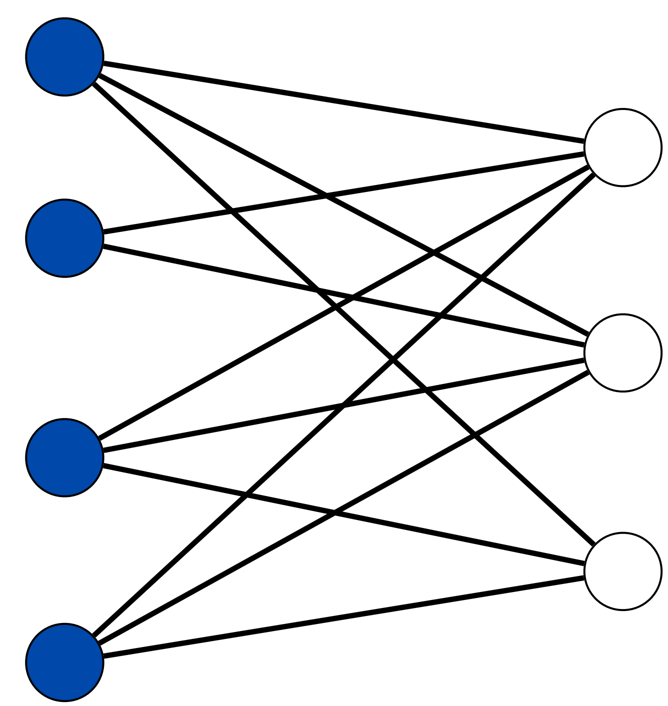
BFS Application: Connectivity

- How to whether a graph is connected using traversals?
    - If the BFS spanning tree contains all nodes of the graph, then the graph is connected
- Suppose the graph is not connected
- How can we find all connected components?
    - Start BFS with any node s, when its done, all nodes in the BFS tree of s are one component
    - Pick another node that is not visited and repeat
    - Number of trees in resulting **forest** is the number of components of the graph

BFS Application: Bipartite Testing

- **Bipartite graph.**
  - An undirected graph is **bipartite** if its nodes can be portioned into two sets S_1, S_2 such that all edges have endpoint in both sets

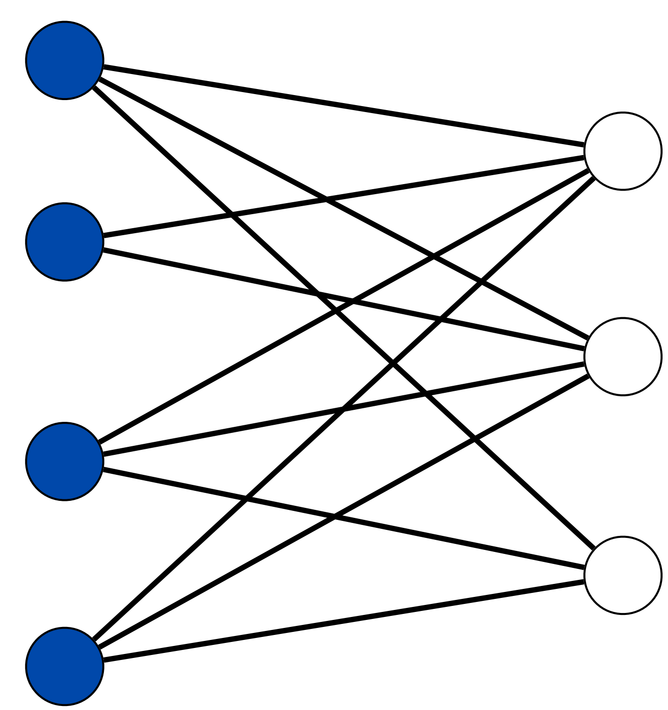
- Models many settings
  - We already encountered an application, which is…?
  - Common in scheduling, one set is machine, other set is jobs



a bipartite graph

BFS Application: Bipartite Testing

- Given a graph $G = (V, E)$ verify if it is bipartite
- Hint: need to use traversals
- But first need to understand structure of bipartite graphs
- **Question:** Can a bipartite graph contain an odd-length cycle?
- How do we prove this?
- In fact, a graph is bipartite if and only if it does not have an odd length cycle
- Let's prove this!



a bipartite graph

Bipartite Testing: Using BFS

Theorem. The following statements are **equivalent** for a connected graph G :

(a) G is bipartite

(b) G has no odd-length cycle

(c) No BFS tree has edges (in G) between vertices at same level

(d) Some BFS tree has no edges (in G) between 2 vertices at same level

Note: Conditions (a) and (b) seem hard to check directly; but conditions (c) and (d) allow an easy check!

Bipartite Testing: Using BFS

Theorem. The following statements are equivalent for a connected graph G :

(a) G is bipartite

(b) G has no odd-length cycle

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between 2 vertices at same level

Proof. (a) ⇒ (b)

Vertices must alternate between V_1 and V_2.

Bipartite Testing: Using BFS

Theorem. The following statements are equivalent for a connected graph G :

(a) G is bipartite

(b) G has no odd-length cycle

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between 2 vertices at same level

Proof. (b) $\Rightarrow$ (c)

Contradiction: Such an edge implies an odd cycle

Bipartite Testing: Using BFS

Theorem. The following statements are equivalent for a connected graph G :

(a) G is bipartite

(b) G has no odd-length cycle

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between 2 vertices at same level

Proof. (c) $\Rightarrow$ (d)

If all BFS trees have a property then some do as well

Bipartite Testing: Using BFS

Theorem. The following statements are equivalent for a connected graph G :

(a) G is bipartite

(b) G has no odd-length cycle

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between 2 vertices at same level

Proof. (d) $\Rightarrow$ (a)

Edges must span consecutive levels: levels provide bipartition of G

Implications of the Theorem

How to check if a graph is bipartite?

- When we visit an edge during BFS, we know the level of both of its endpoints
- So if both ends have the same level, then we can stop ! (G is not bipartite)
- If no such edge is found during traversal, G is bipartite
- Alternate levels give the bipartition

Running time?

- Still $O(n + m)$
- **Certificate.** If G is not bipartite this algorithm gives us a proof of it (the odd cycle that is found)!

Depth-First Search and Directed Graphs

Story So Far

- Breadth-first search
- Using breadth-first search for connectivity
- Using bread-first search for testing bipartiteness

```
BFS (G, s):
 Put s in the queue Q
 While Q is not empty
   Extract v from Q
     If v is unmarked
       Mark v
       For each edge (v, w):
          Put w into the queue Q
```

Generalizing BFS: Whatever-First

If we change how we store the explored vertices (the data structure we use), it changes how we traverse

```
Whatever-First-Search (G, s):
 Put s in the bag
 While bag is not empty
   Extract v from bag
     If v is unmarked
       Mark v
       For each edge (v, w):
          Put w into the bag
```

We can optimize this algorithm by checking whether the node w is marked before we place it the bag.

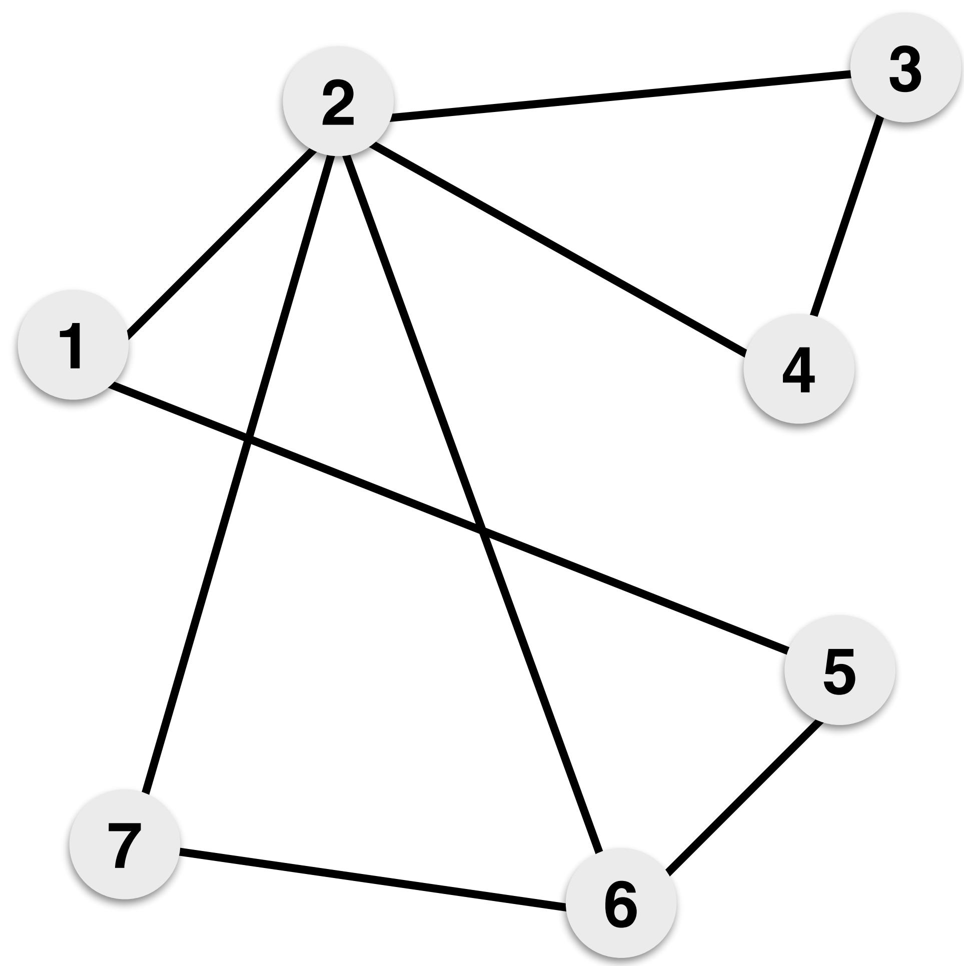
Depth-first search: when bag is a **stack**, not queue

Depth-First Search: Recursive

- Perhaps the most natural traversal algorithm
- Can be written **recursively** as well
- Both versions are the same; can actually see the “recursion stack” in the iterative version

```
Recursive-DFS(u):
   Set status of u to marked # discovered u
   for each edges (u, v):
      if v's status is unmarked:
         DFS(v)
   # done exploring neighbors of u
```

Depth-first Search Example



DFS Running Time

- Inserts and extracts to a stack: $O(1)$ time
- For every node v, explore degree(v) edges
    - $\sum_{v}$ degree(v) $= 2m$
- Connected graphs have $m \geq n - 1$ and thus is $O(m)$ and for general graphs, it is $O(n + m)$

```
ITERATIVEDFS(s):
  PUSH(s)
  while the stack is not empty
      v ← POP
      if v is unmarked
          mark v
          for each edge vw
              PUSH(w)
```

Depth-First Search Tree

- DFS returns a spanning tree, similar to BFS

```
DFS-Tree(G, s):
 Put (∅, s) in the stack S
 While S is not empty
   Extract (p, v) from S
     If v is unmarked
       Mark v
       parent(v) = p
       For each edge (v, w):
          Put (v, w) into the stack S
```

- The spanning tree formed by parent edges in a DFS are usually long and skinny

Depth-First Search Tree

Lemma. For every edge $e = (u, v)$ in G, one of u or v is an ancestor of the other in T.

Proof. Obvious if edge e is in T.

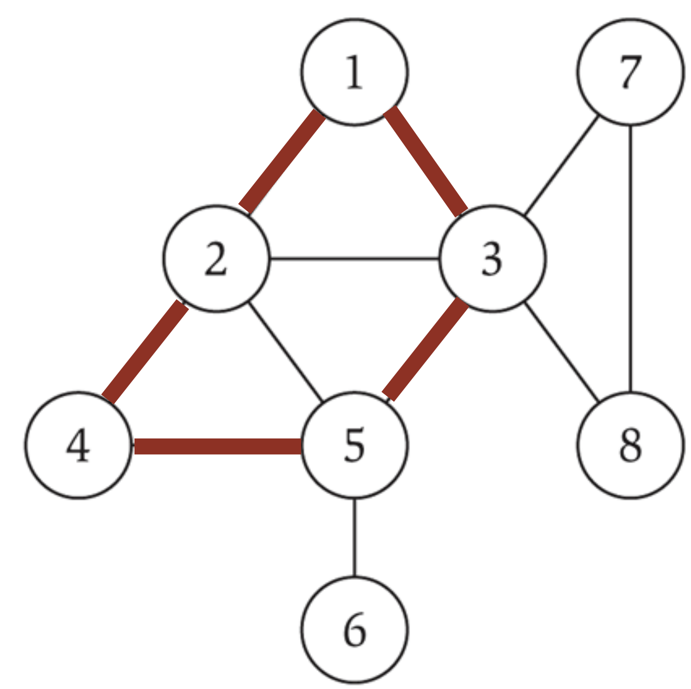
Suppose edge e is not in T. Without loss of generality, suppose DFS is called on u before v.

- When the edge u, v is inspected v must have been already marked visited (why?)
  - Or else $(u, v) \in T$ and we assumed otherwise
- Since $(u, v) \notin T$, v is not marked visited during the DFS call on u
- Must have been marked during a recursive call within DFS(u)
  - Thus v is a descendant of u ∎

Detecting Cycles

Question. Given an undirected connected graph G, how can you detect (in linear time) that contains a cycle?

[Hint. Use DFS]



cycle C = 1-2-4-5-3-1

Detecting Cycles

Question. Given an undirected connected graph G, how can you detect (in linear time) that contains a cycle?

Idea. When we encounter a back edge (u, v) during DFS, that edge is necessarily part of a cycle (cycle formed by following tree edges from u to v and then the back edge from v to u).

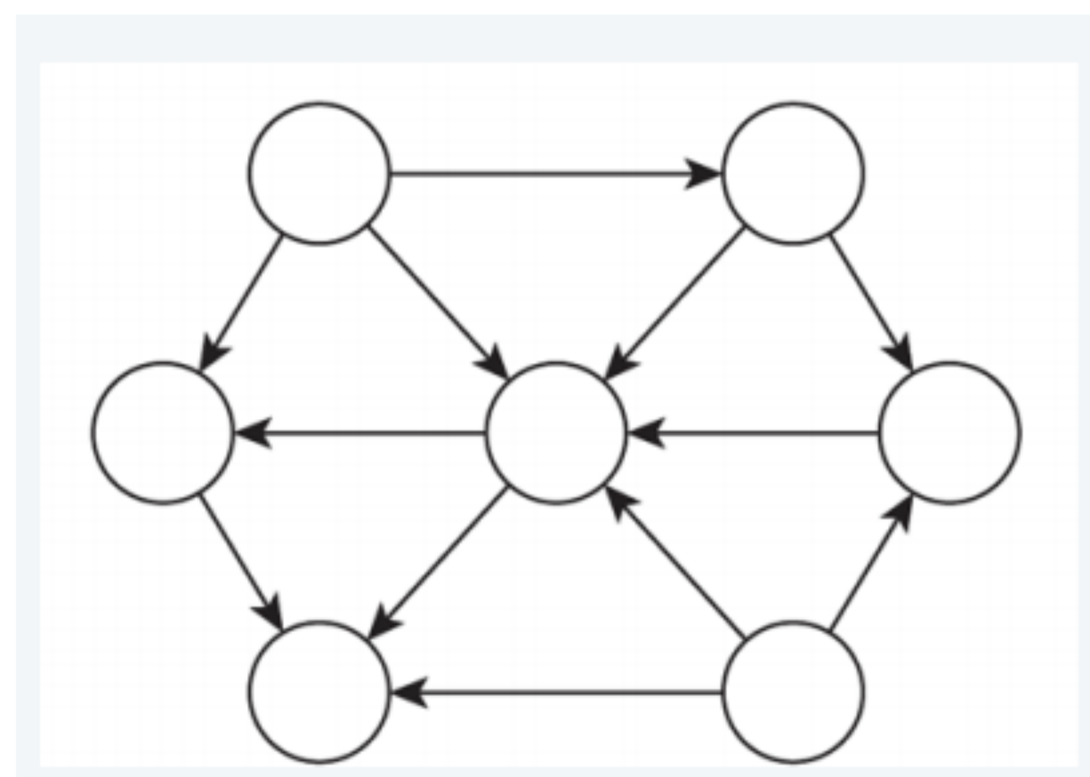
```
Cycle-Detection-DFS(u):
   Set status of u to marked # discovered u
   for each edges (u, v):
      if v's status is unmarked:
         DFS(v)
      else    # found an edge to a marked node
         found a back edge, report a cycle!
   # done exploring neighbors of u
```

Directed Graphs

Notation. $G = (V, E)$.

- Edges have "orientation"
- Edge (u, v) or sometimes denoted $u \to v$, leaves node u and enters node v
- Nodes have "in-degree" and "out-degree"
- No loops or multi-edges (why?)

Terminology of graphs extend to directed graphs: directed paths, cycles, etc.



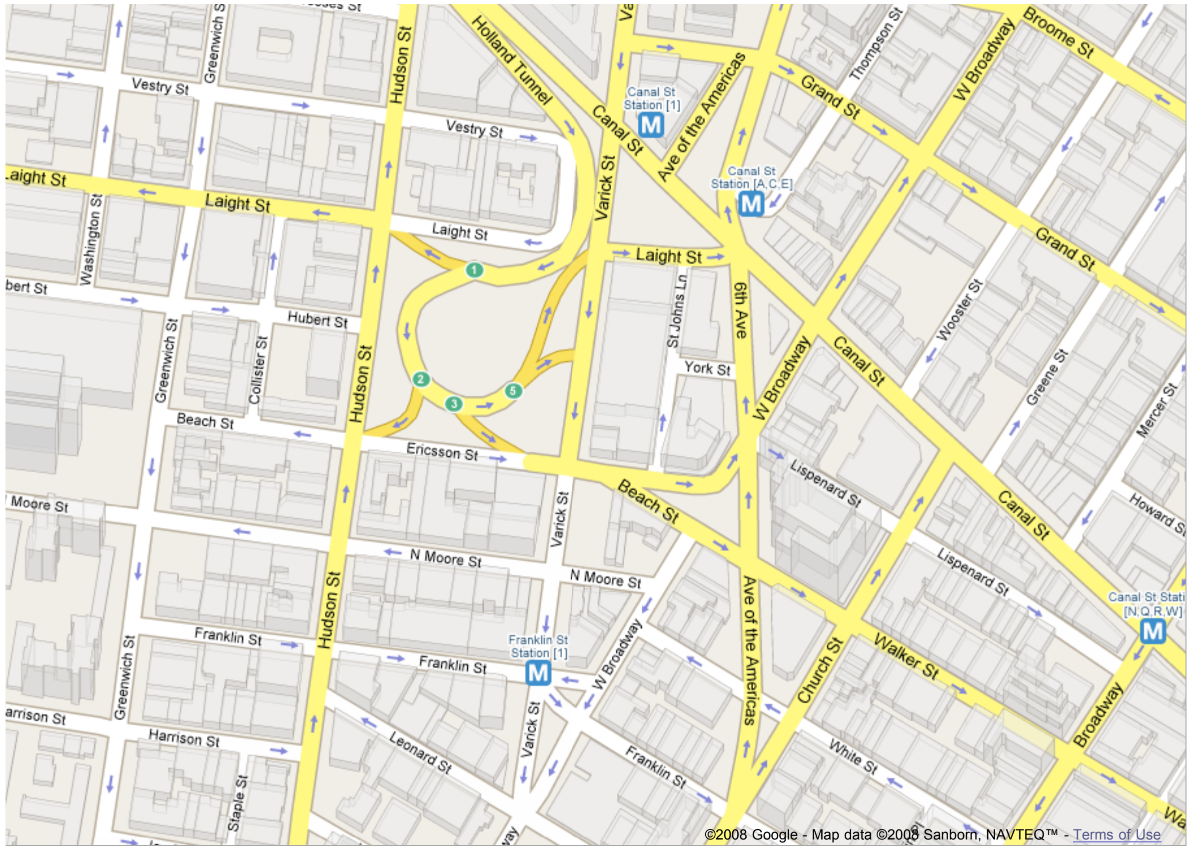
Directed Graphs in Practice

Web graph:

- Webpages are nodes, hyperlinks are edges
- Orientation of edges is crucial
- Search engines use hyperlink structure to rank web pages

Road network

- Road: nodes
- Edge: one-way street



Strong Connectivity & Reachability

Directed reachability. Given a node s find all nodes reachable from s.

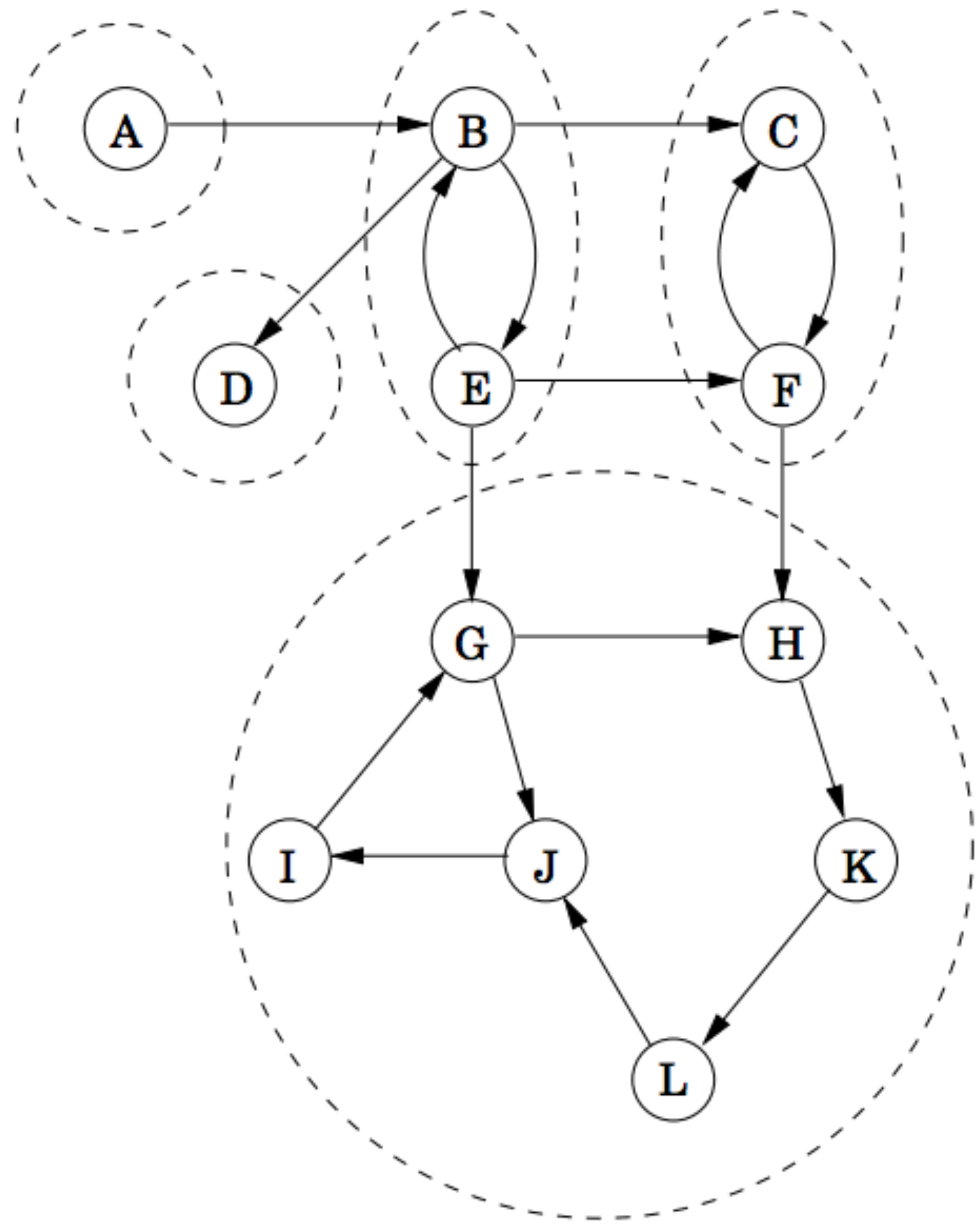
- Can use both BFS and DFS. Both visit exactly the set of nodes reachable from start node s.

- **Strong connectivity.** Connected components in directed graphs defined based on mutual reachability. Two vertices u, v in a directed graph G are mutually reachable if there is a directed path from u to v and from from v to u. A graph G is **strongly connected** if every pair of vertices are mutually reachable

- The mutual reachability relation decomposes the graph into strongly-connected components

- **Strongly-connected components.** For each $v \in V$, the set of vertices mutually reachable from v, defines the strongly-connected component of G containing v.

Strongly Connected Components



Deciding Strongly Connected

First idea. How can we use BFS/DFS to determine strong connectivity? Recall: BFS/DFS on graph G starting at v will identifies all vertices reachable from v by directed paths

- Pick a vertex v. Check to see whether every other vertex is reachable from v;
- Now see whether v is reachable from every other vertex

Analysis

- First step: one call to BFS: $O(n + m)$ time
- Second step: $n - 1$ calls to BFS: $O(n(n + m))$ time
- **Can we do better?**

Testing Strong Connectivity

Idea. Flip the edges of G and do a BFS on the new graph

- Build $G_{\text{rev}} = (V, E_{\text{rev}})$ where $(u, v) \in E_{\text{rev}}$ iff $(v, u) \in E$
- There is a directed path from v to u in G_{rev} iff there is a directed path from u to v in G
- Call BFS(G_{rev}, v): Every vertex is reachable from v (in G_{rev}) if and only if v is reachable from every vertex (in G).

Analysis (Performance)

- BFS(G, v): $O(n + m)$ time
- Build G_{rev}: $O(n + m)$ time. [Do you believe this?]
- BFS(G_{rev}, v): $O(n + m)$ time
- Overall, linear time algorithm!

Kosaraju's Algorithm

Testing Strong Connectivity

Idea. Flip the edges of G and do a BFS on the new graph

- Build $G_{\text{rev}} = (V, E_{\text{rev}})$ where $(u, v) \in E_{\text{rev}}$ iff $(v, u) \in E$
- There is a directed path from v to u in G_{rev} iff there is a directed path from u to v in G
- Call BFS(G_{rev}, v): Every vertex is reachable from v (in G_{rev}) if and only if v is reachable from every vertex (in G).

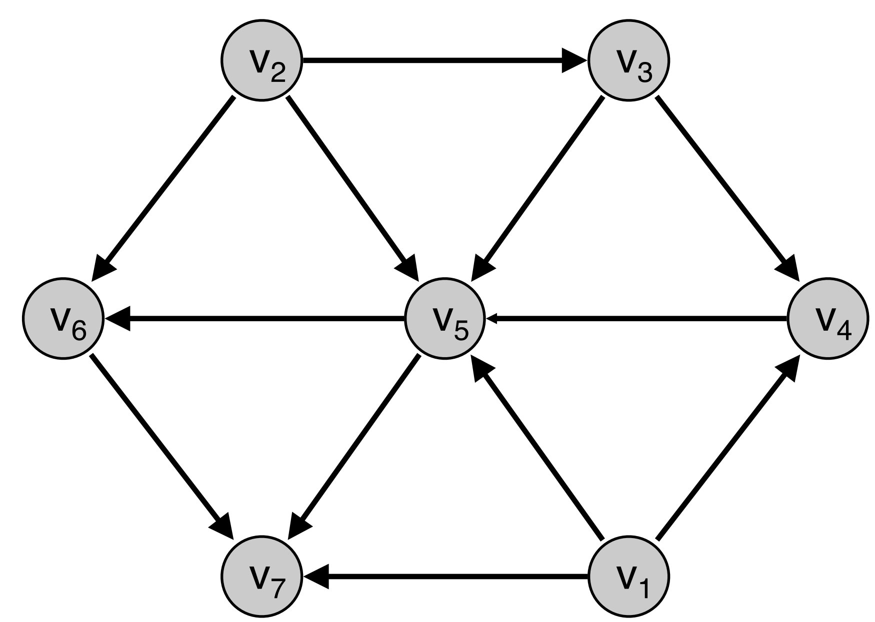
Analysis (Correctness)

- **Claim.** If v is reachable from every node in G and every node in G is reachable from v then G must be strongly connected
- **Proof.** For any two nodes $x, y \in V$, they are mutually reachable through v, that is, $x \rightsquigarrow v \rightsquigarrow y$ and $y \rightsquigarrow v \rightsquigarrow z$ ∎

Directed Acyclic Graphs (DAGs)

Definition. A directed graph is acyclic (or a DAG) if it contains no (directed) cycles.

Question. Given a directed graph G, can you detect if it has a cycle in linear time? Can we apply the same strategy (DFS) as we did for undirected graphs?

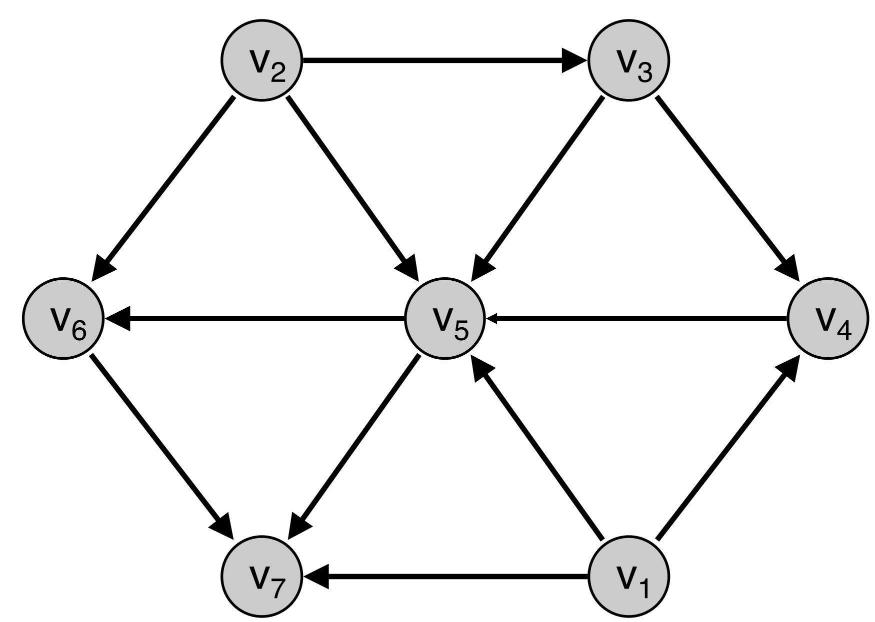


a DAG

Directed Acyclic Graphs (DAGs)

Definition. A directed graph is acyclic (or a DAG) if it contains no (directed) cycles.

Question. Given a directed graph G, can you detect if it has a cycle in linear time? Can we apply the same strategy (DFS) as we did for undirected graphs?



a DAG

Directed Acyclic Graphs (DAGs)

Definition. A directed graph is acyclic (or a DAG) if it contains no (directed) cycles.

Question. Given a directed graph G, can you detect if it has a cycle in linear time? Can we apply the same strategy (DFS) as we did for undirected graphs?

```
Cycle-Detection-Directed-DFS(u):
   Set status of u to marked # discovered u
   for each edges (u, v):
      if v's status is unmarked:
         DFS(v)
      else if v is marked but not finished
         report a cycle!
   mark u finished
   # done exploring neighbors of u
```