Graphs (Continued)

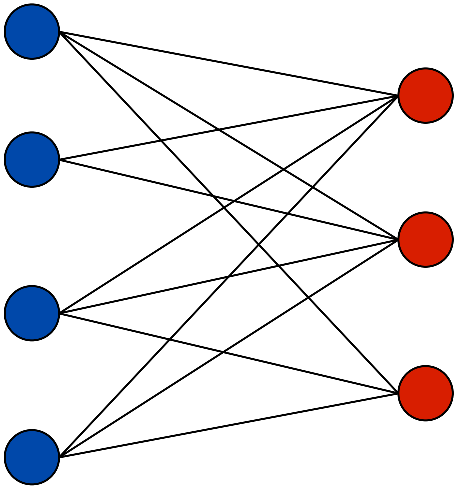
Testing Bipartiteness: A Breadth First Search Application

Bipartite Graphs

Def. An undirected graph G = (V, E) is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

Applications.

- Stable marriage
- Professors : courses
- Scheduling: machines = red, jobs = blue.

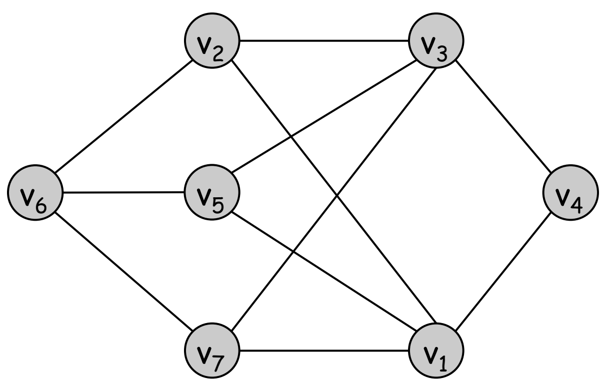


a bipartite graph

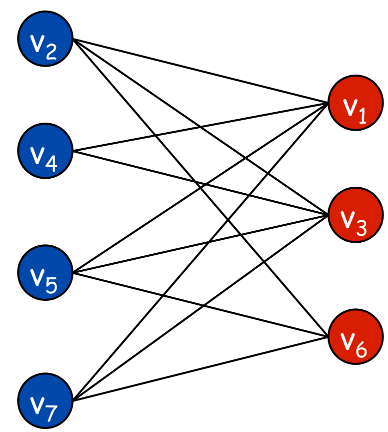
Testing Bipartiteness

Testing bipartiteness. Given a graph G, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



a bipartite graph G

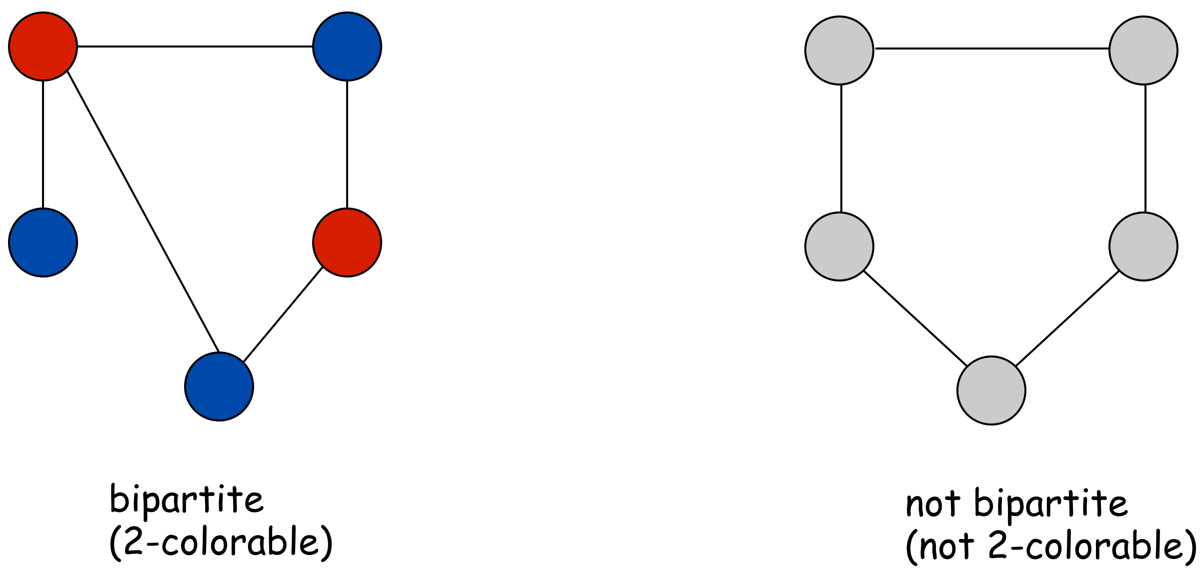


another drawing of G

An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

Pf. Not possible to 2-color the odd cycle, let alone G.

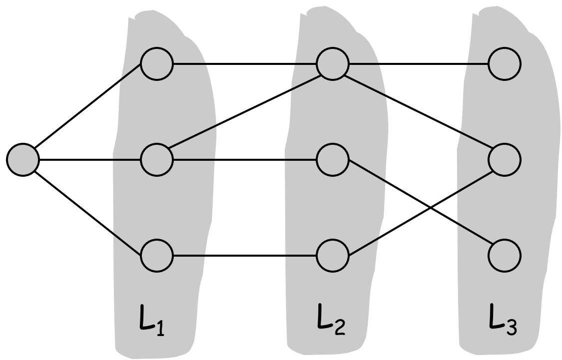


Bipartite Graphs

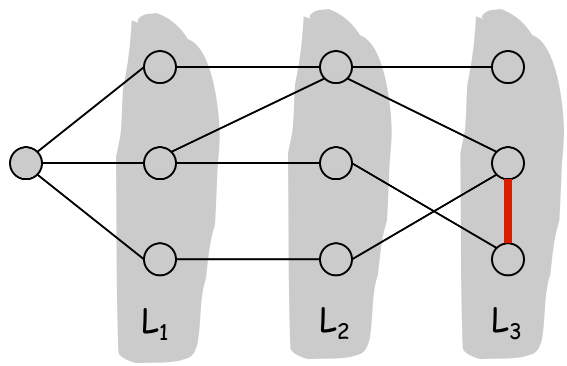
Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



Case (i)



Case (ii)

Bipartite Graphs

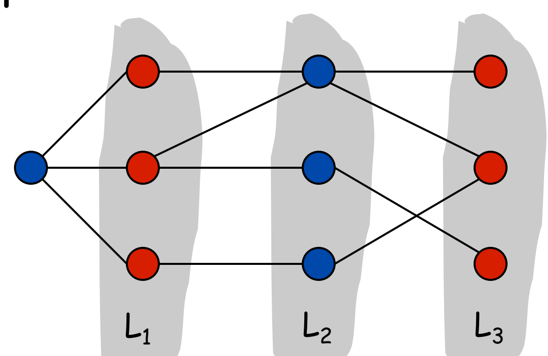
Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (i)

- Suppose no edge joins two nodes in the same layer.
- Previous result – all edges connect nodes no more than one layer apart, so this implies all edges join nodes on adjacent levels.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.



Case (i)

Bipartite Graphs

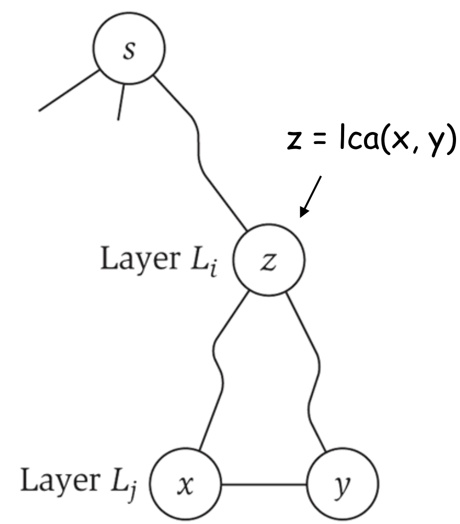
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

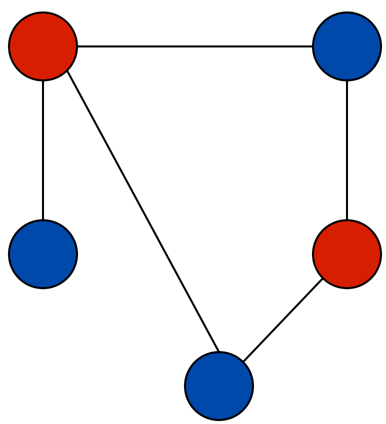
Pf. (ii)

- Suppose (x, y) is an edge with x, y in same level L_j.
- Let $z = \text{lca}(x, y)$ = lowest common ancestor.
- Let L_i be level containing z.
- Consider cycle that takes edge from x to y, then path from y to z, then path from z to x.
- Its length is $\underbrace{1}_{(x,\,y)} + \underbrace{(j-i)}_{\text{path from } y \text{ to } z} + \underbrace{(j-i)}_{\text{path from } z \text{ to } x}$, which is odd. ▪

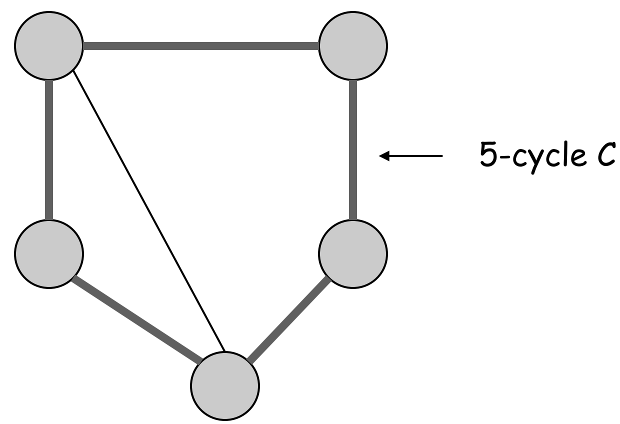


Obstruction to Bipartiteness

Corollary. A graph G is bipartite iff it contain no odd length cycle.



bipartite
(2-colorable)

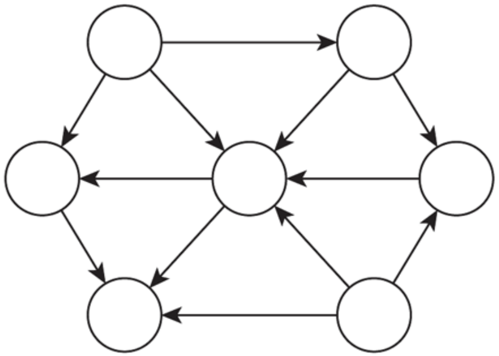


not bipartite
(not 2-colorable)

Directed Acyclic Graphs (DAG) and Topological Ordering

Directed Graphs

Directed graph. $G = (V, E)$
- Edge (u, v) goes from node u to node v.



Ex. Web graph - hyperlink points from one web page to another.
- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.

Directed Graph Search

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

Directed s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Graph search. BFS (DFS too) extends naturally to directed graphs.

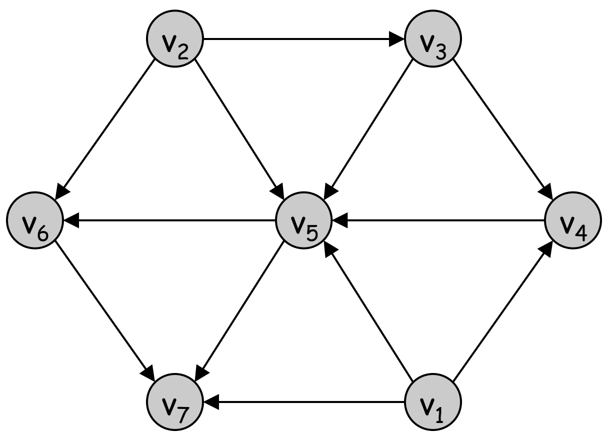
Web crawler. Start from web page s. Find all web pages linked from s, either directly or indirectly.

Directed Acyclic Graphs

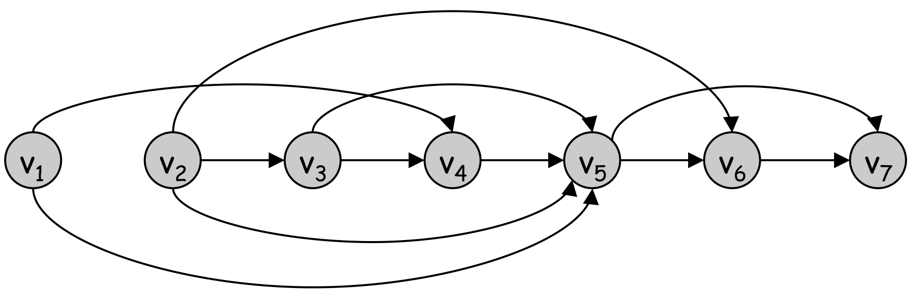
Def. A DAG is a directed graph that contains no directed cycles.

Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.

Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



a DAG



a topological ordering

Precedence Constraints

Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j.

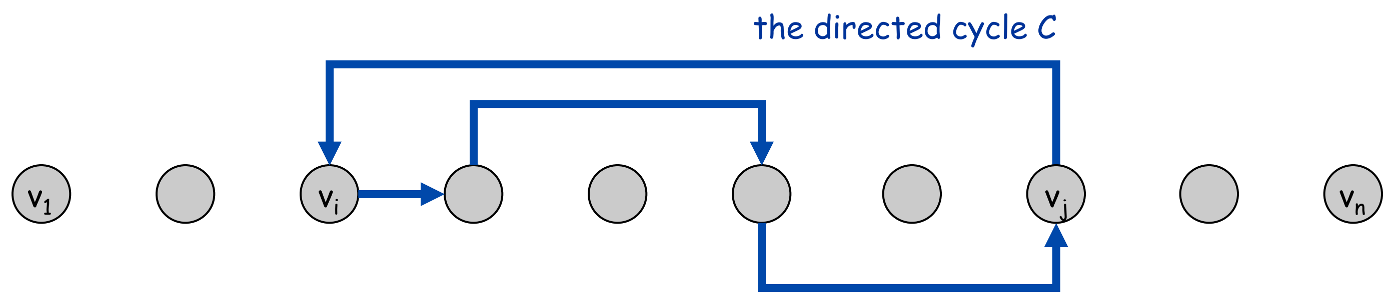
Applications.

- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j. Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

Lemma. If G has a topological order, then G is a DAG.

Pf. (by contradiction)

- Suppose that G has a topological order $v_1, \ldots, v_n$ and that G also has a directed cycle C. Let's see what happens.
- Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
- By our choice of i, we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪



the supposed topological order: $v_1, \ldots, v_n$

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

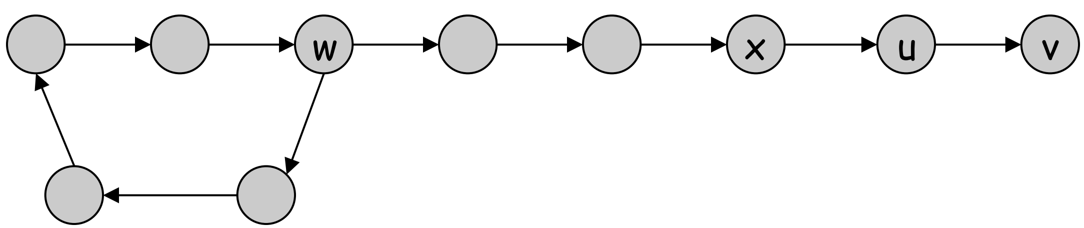
Q. If so, how do we compute one?

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.
- Then, since u has at least one incoming edge (x, u), we can walk backward to x.
- Repeat until we visit a node, say w, twice.
- Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle. ▪



Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a topological ordering.

Pf. (by induction on n)

- Base case: true if n = 1.
- Given DAG on n > 1 nodes, find a node v with no incoming edges.
- G - { v } is a DAG, since deleting v cannot create cycles.
- By inductive hypothesis, G - { v } has a topological ordering.
- Place v first in topological ordering; then append nodes of G - { v }
- in topological order. This is valid since v has no incoming edges. ▪

To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of $G-\{v\}$
  and append this order after v

Complexity: $O(n^2)$ for basic algorithm

Topological Sorting Algorithm: Running Time

Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Pf.

- Maintain the following information:
  - `count[w]` = remaining number of incoming edges
  - S = set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete v
  - remove v from S
  - decrement `count[w]` for all edges from v to w, and add w to S if `count[w]` hits 0
  - this is $O(1)$ per edge ▪

Topological Sort Examples

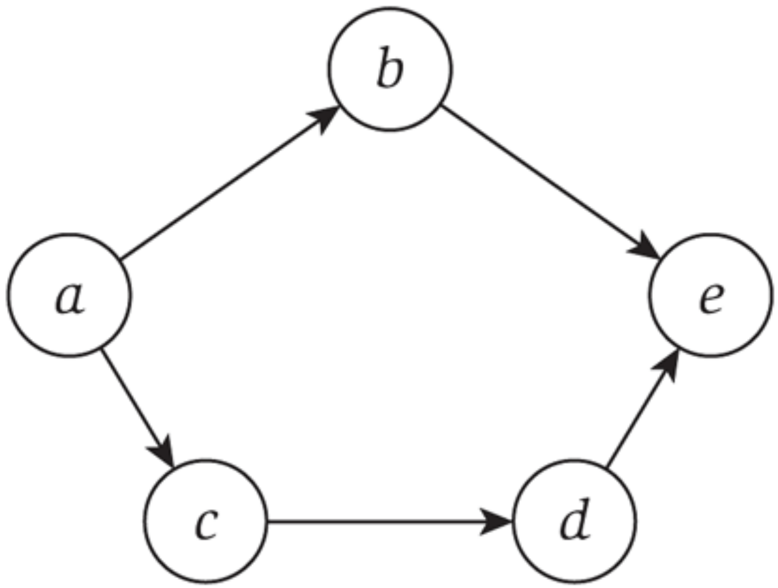


Figure 3.9 How many topological orderings does this graph have?

Topological Sort Examples

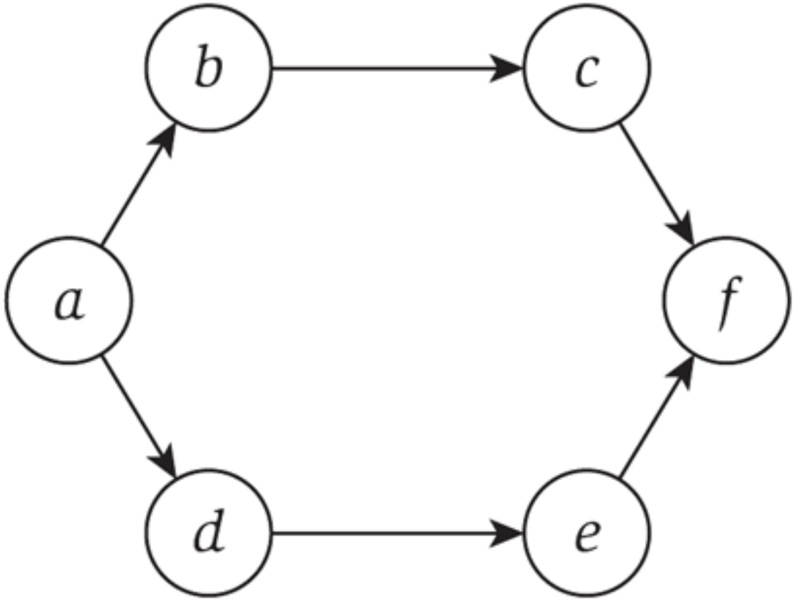


Figure 3.10 How many topological orderings does this graph have?

Topological Ordering – A Different Approach

Idea: Use recursive DFS in which we keep track of when each vertex is "finished"

- A vertex is finished when all of its outgoing edges have been explored and we are moving up the tree from that vertex
- Start at any random vertex
  - Perform a recursive DFS from that vertex
  - Store finishing time (finishing order) of all vertices that are visited as part of that DFS (counting up from 1)
- Repeat as necessary for remaining vertices (continue counting where we left off)

- Topological ordering is a listing of the vertices in decreasing finishing time

- (NOTE – still requires a DAG to generate a topological ordering. You can run the algorithm on a graph with cycles, but the result will have at least one edge pointed in the wrong direction

Topological Ordering – A Different Approach

Complexity: $O(m + n)$

```
TopOrder ( G=(V,E) )
for every vertex v
    seen[v]=false
    fin[v]= ∞
time=0
for every vertex s
    if not seen[s] then
        DFS(s)

DFS(v)
seen[v]=true
for every neighbor u of v
    if not seen[u] then
        DFS(u)
time++
fin[v]=time (and output v)
```

- Each vertex is visited once: $O(n)$ across entire run-time
- Each edge is considered once: $O(m)$ across entire run-time
- Computing finish time is done once for each vertex: $O(n)$ across entire run-time

Connectivity in Directed Graphs

Strong Connectivity

Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.

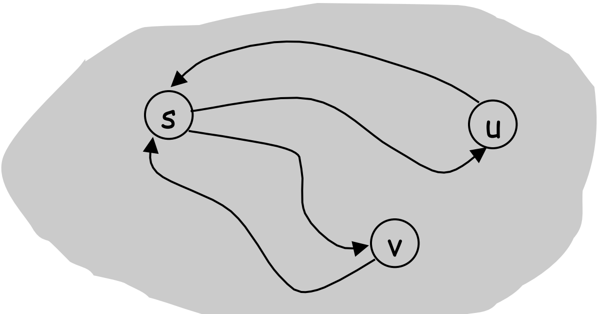
Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Pf. $\Rightarrow$ Follows from definition.

Pf. $\Leftarrow$ Path from u to v: concatenate u-s path with s-v path.

Path from v to u: concatenate v-s path with s-u path. ▪

ok if paths overlap



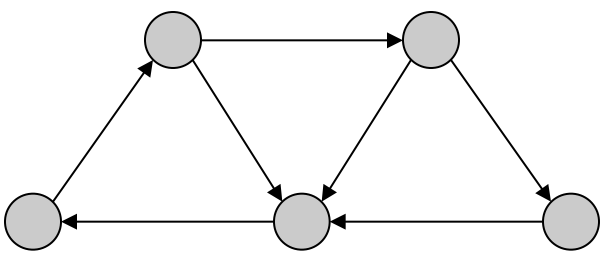
Strong Connectivity: Algorithm

Theorem. Can determine if G is strongly connected in $O(m + n)$ time.
Pf.

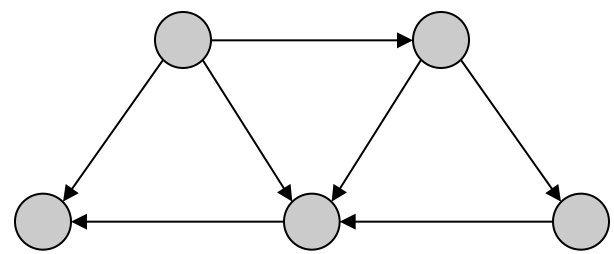
- Pick any node s.
- Run BFS from s in G.
- Run BFS from s in G^{rev}.

reverse orientation of every edge in G

- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. ▪



strongly connected



not strongly connected

Finding all Strongly Connected Components

First Approach: Use DFS n times to determine the set of reachable vertices for each vertex. Then go through the sets to see which vertices are in each other's sets.

- Each DFS is $O(n+m)$, so this step is $O(n(n+m))$
- Checking for mutual reachability is $O(n^2)$. Consider storing reachability information as an adjacency matrix and then just accessing elements (only need to access each element at most once).

Finding all Strongly Connected Components

Idea: Use DFS in which we keep track of when each vertex is "finished" to determine the strongly connected components of a directed graph

- Do a topological ordering using DFS with finish times for the graph G
- Compute G^T by reversing all edges of G
- Consider the vertices in decreasing order of finishing time for a DFS using G^T.
- All vertices that are visited in a particular DFS search on G^T together comprise a strongly connected component.
- Repeated DFS searches / strongly connected components are formed until all vertices have been visited.

- Let's see an example

Strongly Connected Components

Claim: Finding all Strongly Connected Components runs in O(m+n) time

```
STRONGLY-CONNECTED COMPONENTS ( G=(V,E) )
1.   for every vertex v
2.      seen[v]=false
3.      fin[v]= ∞
4.   time=0
5.   for every vertex s
6.      if not seen[s] then
7.         DFS(G,s) (the finished-time version)
8.   compute G^T by reversing all edges of G
9.   process vertices by decreasing finished time
10.  seen[v]=false for every vertex v
11.  for every vertex v do
12.     if not seen[v] then
13.        output vertices seen by DFS(v)
```

- 1-7 are top. order $O(m+n)$
- 8 is $O(m + n)$
- 9 is $O(1)$ already computed
- 10-13 are $O(m+n)$ DFS