CHAPTER 3
GRAPHS

Iris Hui-Ru Jiang
Fall 2017

Department of Electrical Engineering
National Taiwan University

Outline

- Content:
  - Basic definitions and applications
  - Graph connectivity and graph traversal
  - Implementation
  - Testing bipartiteness: an application of BFS
  - Connectivity in directed graphs
  - Directed acyclic graphs and topological ordering
- Reading:
  - Chapter 3

Keys to Success: CAR Theorem

- **Chang's CAR Theorem**

Criticality

Abstraction

Restriction

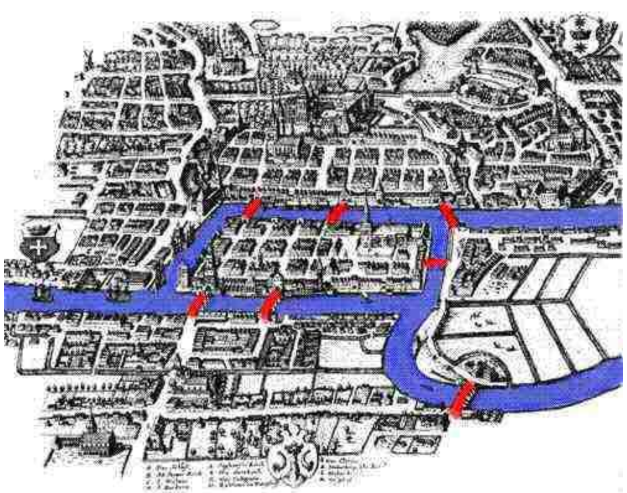
- **Recap stable matching**
- Extract the essence
  - Identify the clean core
  - Remove extraneous detail
- Represent in an abstract form
  - First think at high-level
    - Devise the algorithm
  - Then go down to low-level
    - Complete implementation
- Simplify unimportant things
  - List the limitations
  - Show how to extend

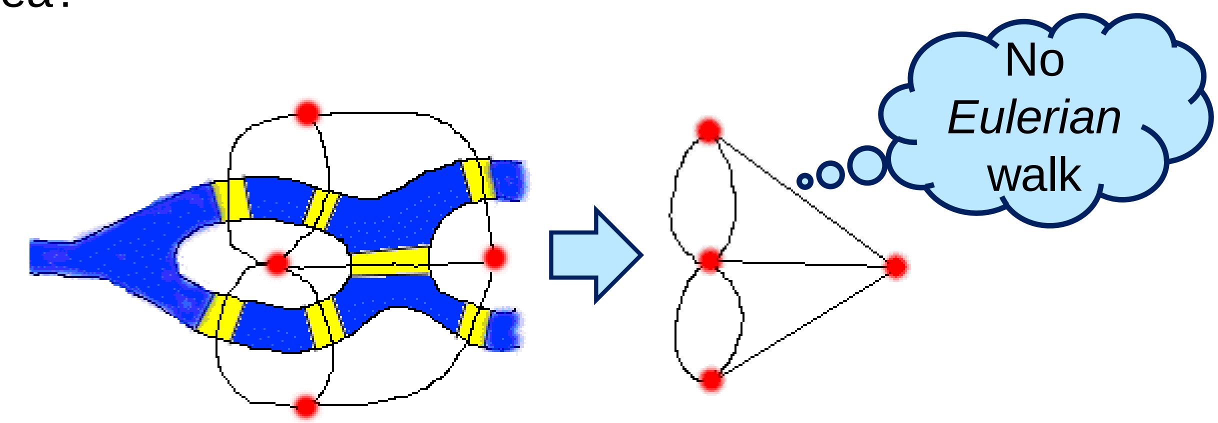
Basics

Definitions and applications

Salute to Euler!

- Our focus in this course is on problems with a discrete flavor.
- One of the most fundamental and expressive of combinatorial structures is the graph.
  - Invented by L. Euler based on his proof on the Königsberg bridge problem (the seven bridge problem) in 1736.
    - Is it possible to walk across all the bridges exactly once and return to the starting land area?



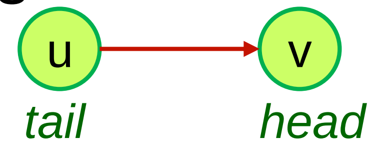


L. Euler, Solutioproblematis ad geometriamsituspertinentis, *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, Vol. 8, pp. 128—140, 1736 (published 1741).

Graph

- A graph encodes pairwise relationships among objects.
- A graph $G = (V, E)$ consists of
  - A collection V of nodes (a.k.a. vertices)
  - A collection E of edges
    - Each edge joins two nodes
    - $e = \{u, v\} \in E$ for some $u, v \in V$
- In an undirected graph: symmetric relationships
  - Edges are undirected, i.e., $\{u, v\} == \{v, u\}$
    - e.g., *u* and *v* are family.

- In a directed graph: asymmetric relationships
  - Edges are directed, i.e., $(u, v) != (v, u)$
    - e.g., *u* knows *v* (celebrity), while *v* doesn't know *u.*



- *v* is one of *u*'s neighbor if there is an edge (u, v)
  - Adjacency

Examples of Graphs (1/6)

- It's useful to digest the meaning of the nodes and the meaning of the edges in the following examples.
  - It's not important to remember them.
- Transportation networks:

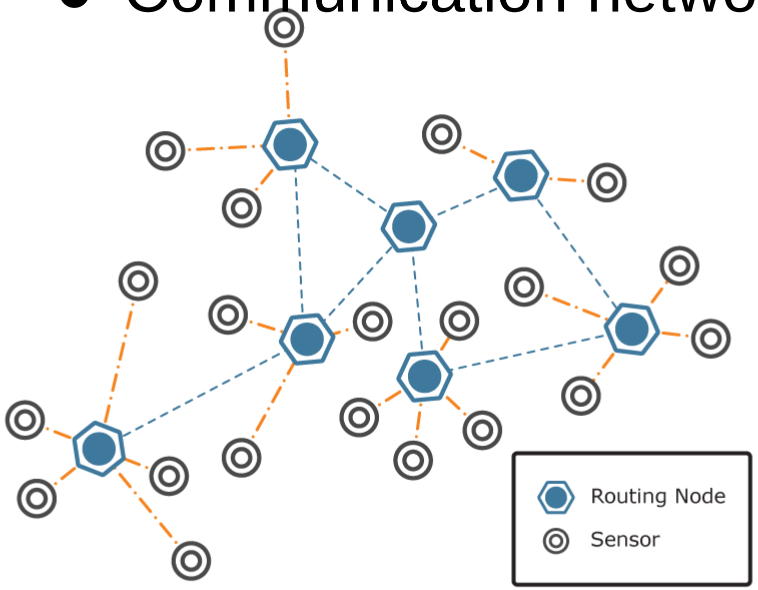
China Airlines international route map
(node: city; edge: non-stop flight)

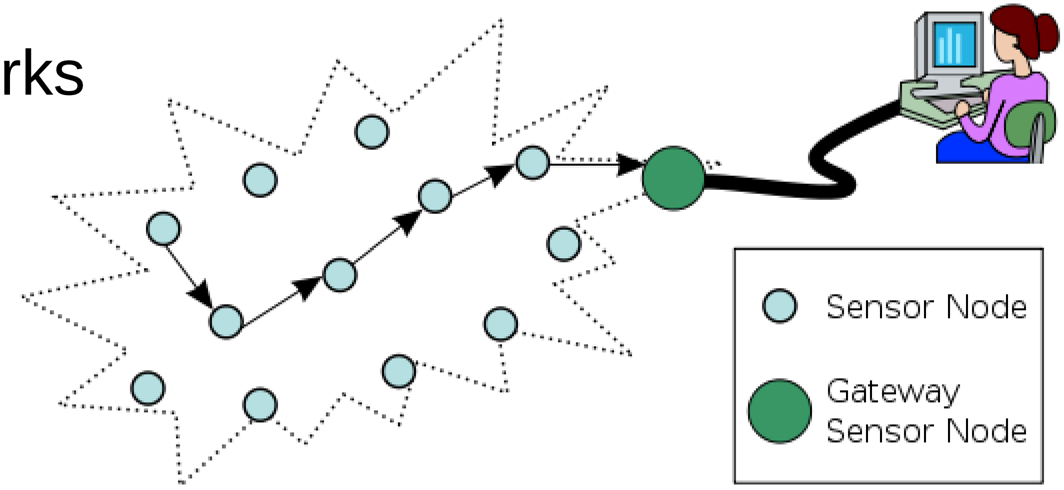
London underground map
(node: station; edge: adjacent stations)

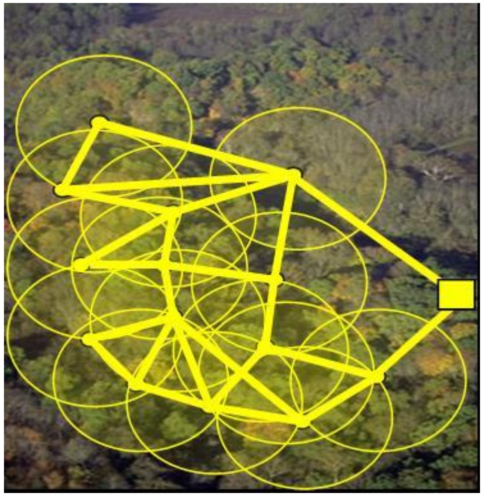
不敗經典設計—倫敦地鐵地圖 (H. Beck, 1933)

Examples of Graphs (2/6)

- Communication networks



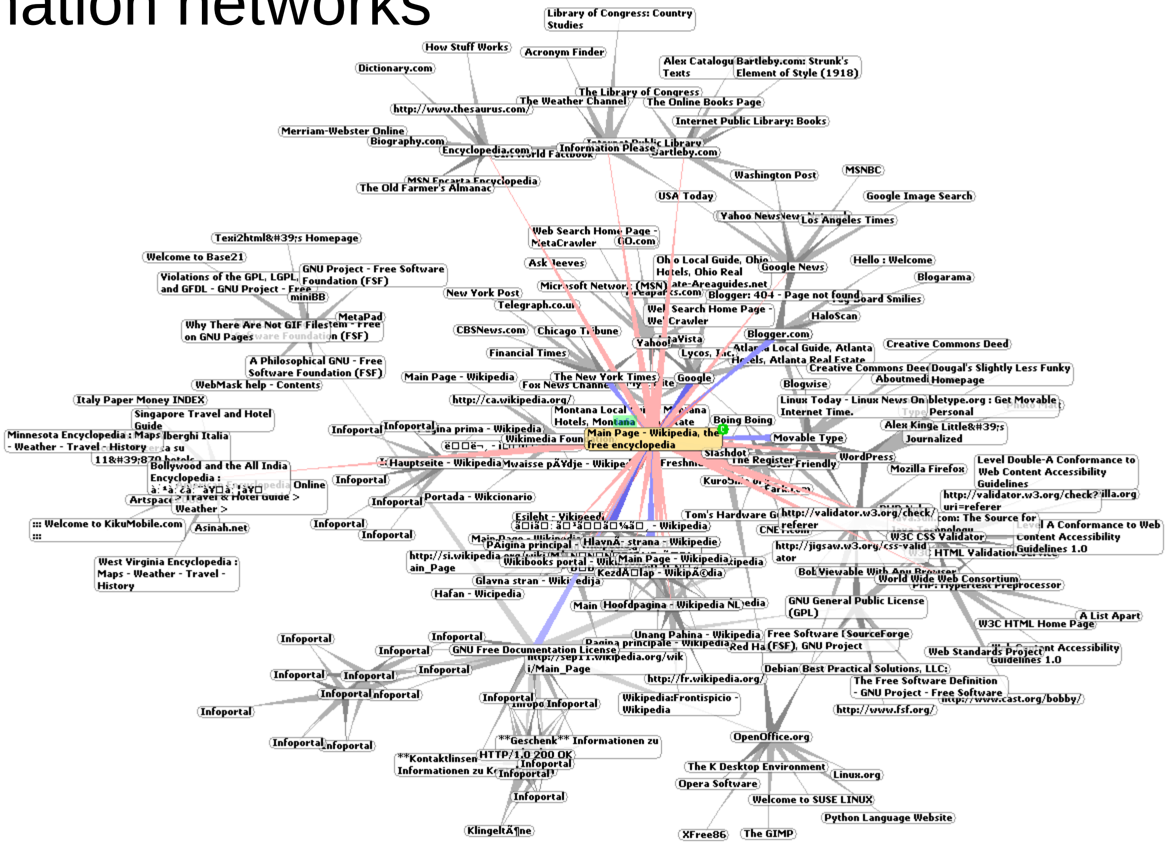




Wireless sensor network
(node: sensor; edge: signal broadcasting)

Examples of Graphs (3/6)

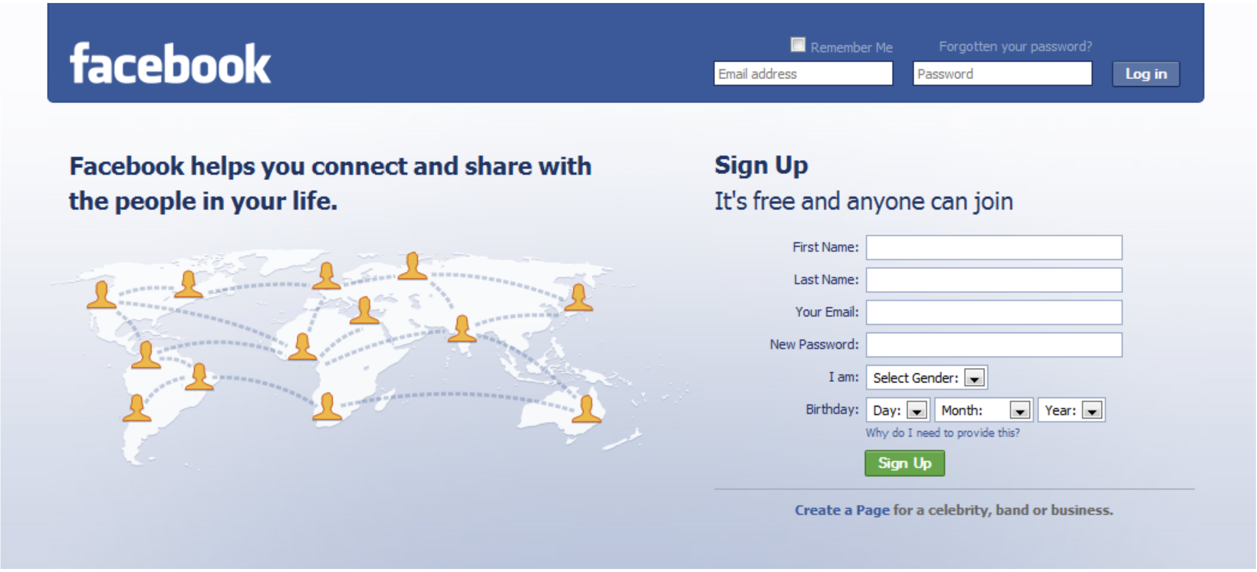
- Information networks



World Wide Web
(node: webpage; edge: hyperlink)

Examples of Graphs (4/6)

- Social networks
- Six degrees of separation
  - All living things and everything else in the world are six or fewer steps away from each other

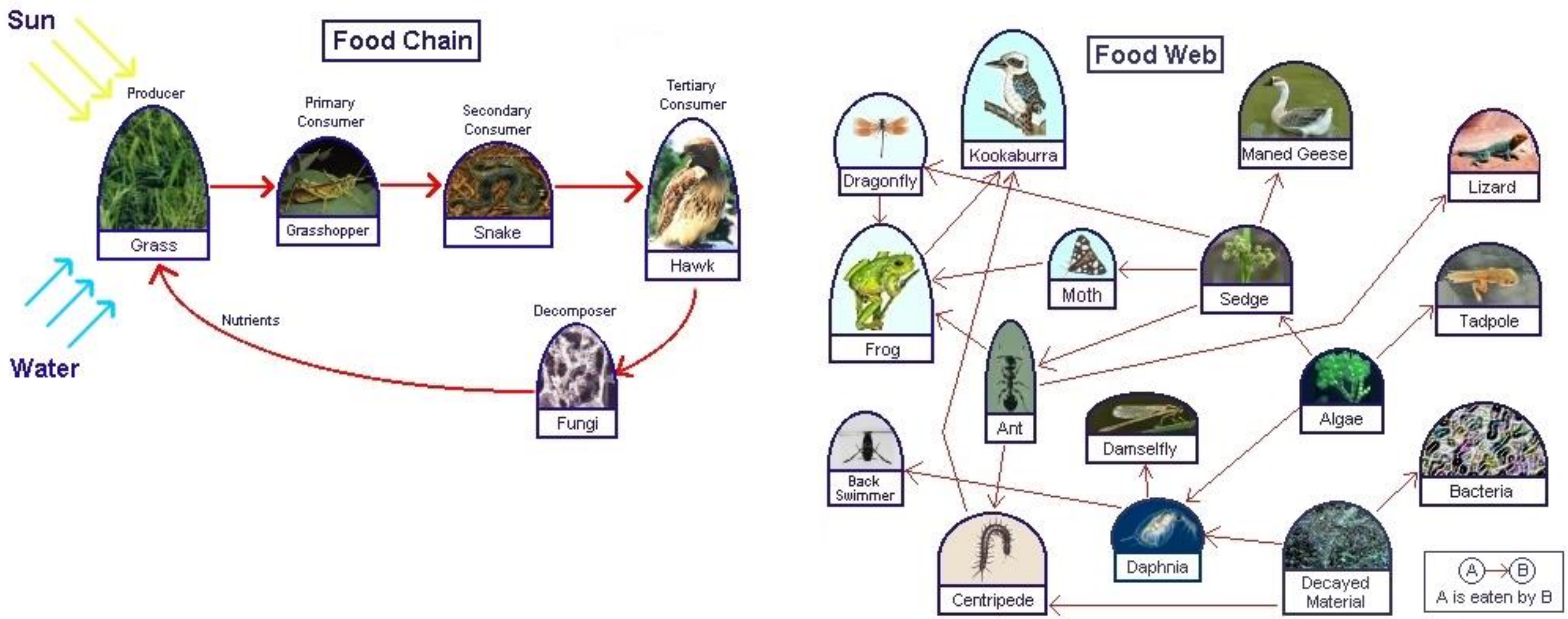


Facebook
(node: people; edge: friendship)

https://www.facebook.com/

Examples of Graphs (5/6)

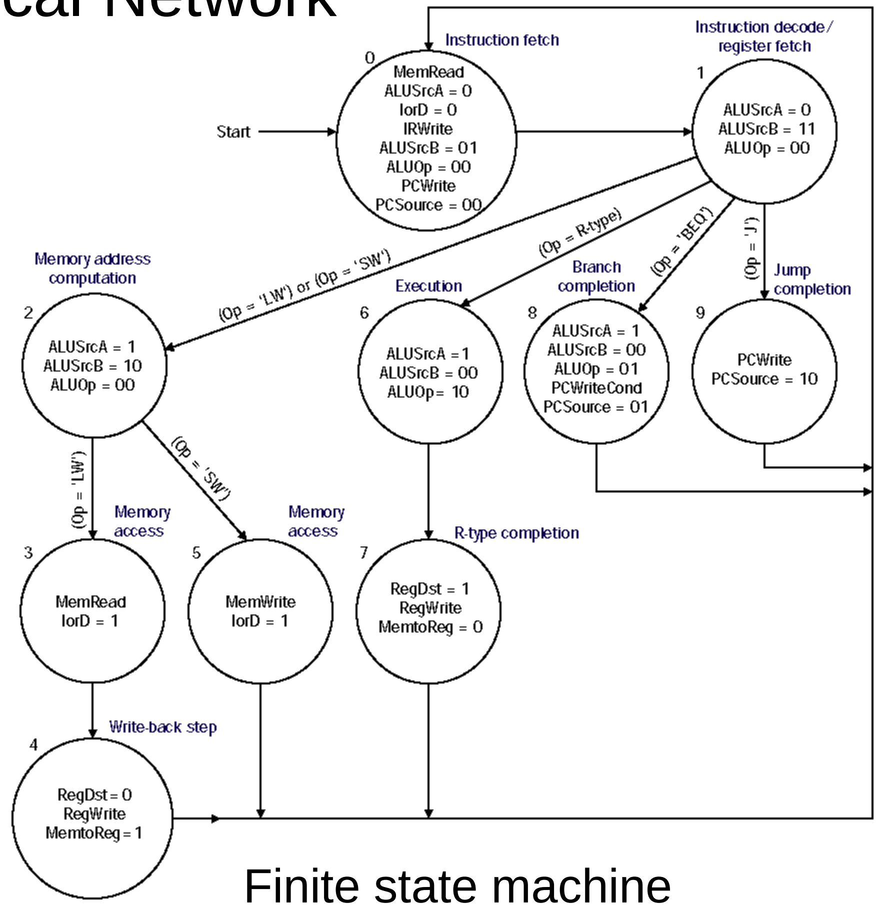
- Dependency networks



Food chain/web
(node: species; edge: from prey to predator)

Examples of Graphs (6/6)

- Technological Network

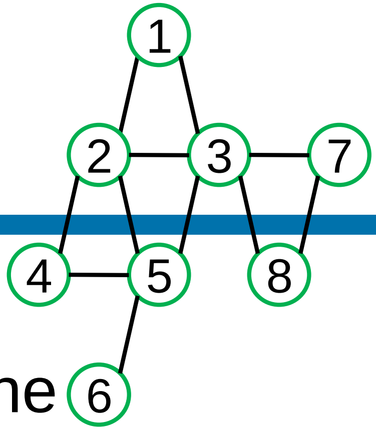


Finite state machine
(node: state; edge: state transition)

Paths and Connectivity (1/2)

- One of the fundamental operations in a graph is that of traversing a sequence of nodes connected by edges.
  - Browse Web pages by following hyperlinks
  - Join a 10-day tour from Taipei to Europe on a sequence of flights
  - Pass gossip by word of mouth (by message of mobile phone) from you to someone far away

Paths and Connectivity (2/2)



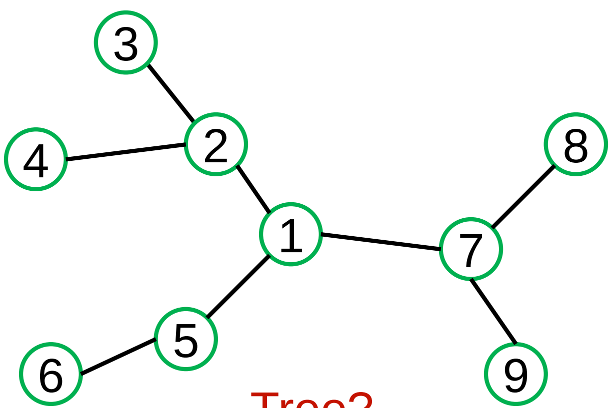
- A path in an undirected graph $G = (V, E)$ is
- a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E.
- A path is simple if all nodes are distinct.
- A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first k-1 nodes are all distinct.
- An undirected graph is connected if, for every pair of nodes u and v, there is a path from u to v.
- The distance between nodes u and v is the minimum number of edges in a u-v path. (∞ for disconnected)
- Note: These definitions carry over naturally to directed graphs with respect to the directionality of edges.

Path P = 1, 2, 4, 5, 3, 7, 8
Cycle C = 1, 2, 4, 5, 3, 1

Trees

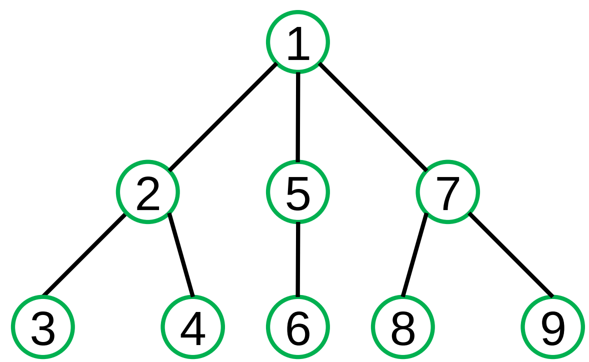
- An undirected graph is a tree if it is connected and does not contain a cycle.
  - Trees are the simplest kind of connected graph: deleting any edge will disconnect it.
- Thm: Let G be an undirected graph on n nodes. Any two of the following statements imply the third.
  - G is connected.
  - G does not contain a cycle.
  - G has n-1 edges.

G is a tree if it satisfies any two of the three statements



Tree?

Grab 1 and let the rest hang downward

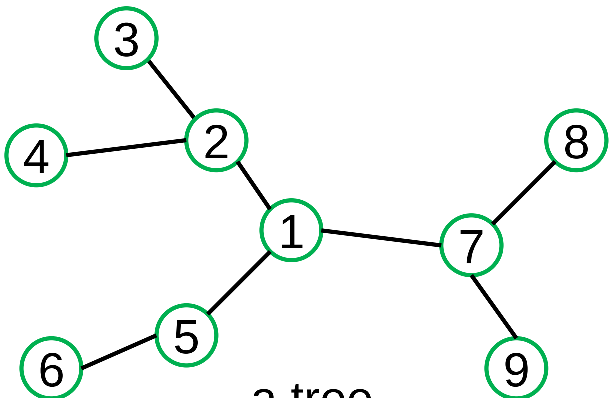


Yes!

Rooted Trees

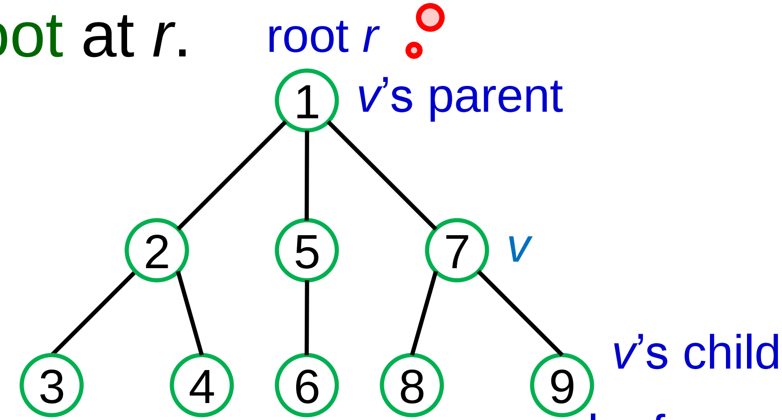
Directed edges: asymmetric relationship

- A rooted tree is a tree with its root at *r*.



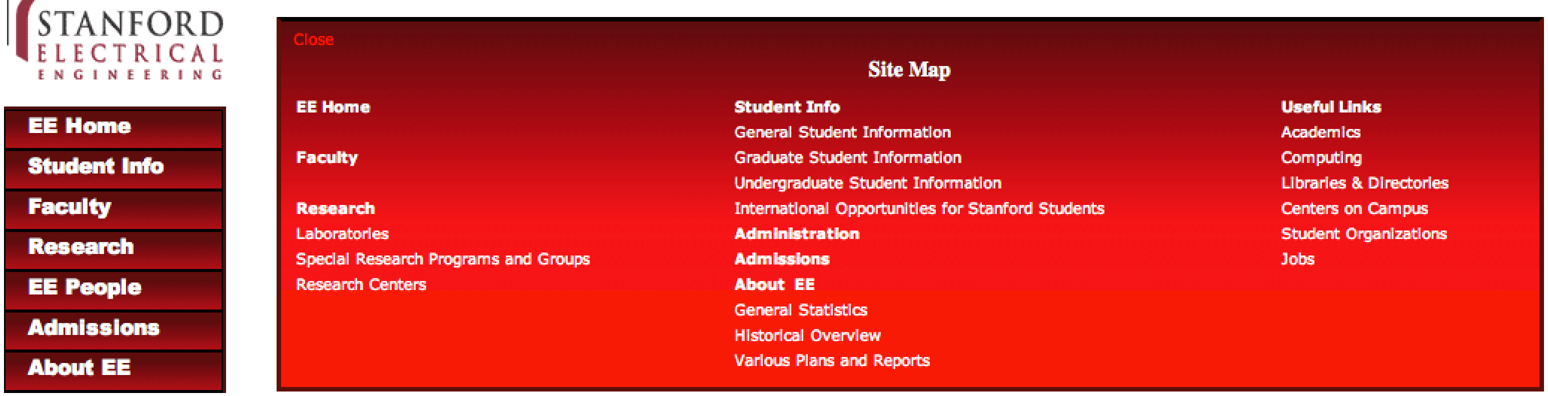
a tree

Grab 1 and let the rest hang downward



the same tree, rooted at 1

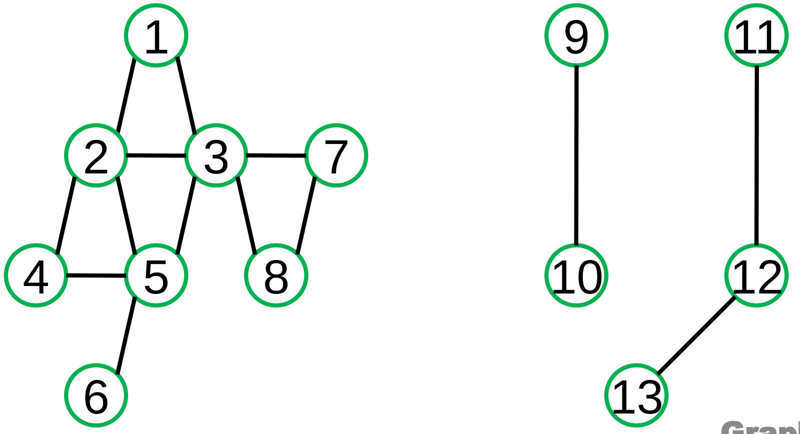
- Rooted trees encode the notion of a hierarchy.
  - e.g., sitemap of a Web site
    - The tree-like structure facilitates navigation (root: entry page)



Graph Connectivity and Graph Traversal

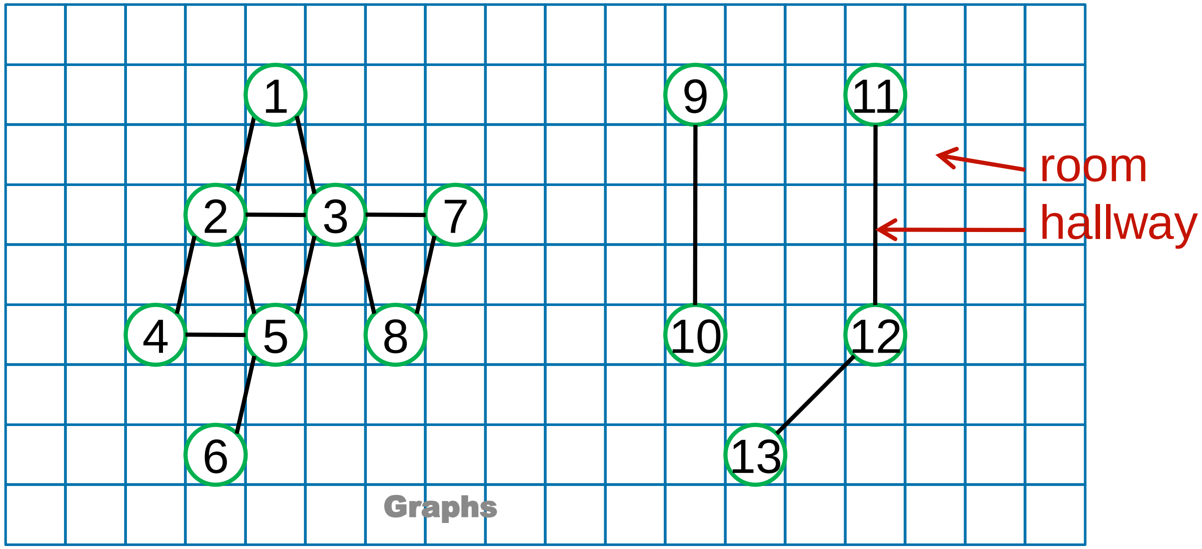
BFS

DFS



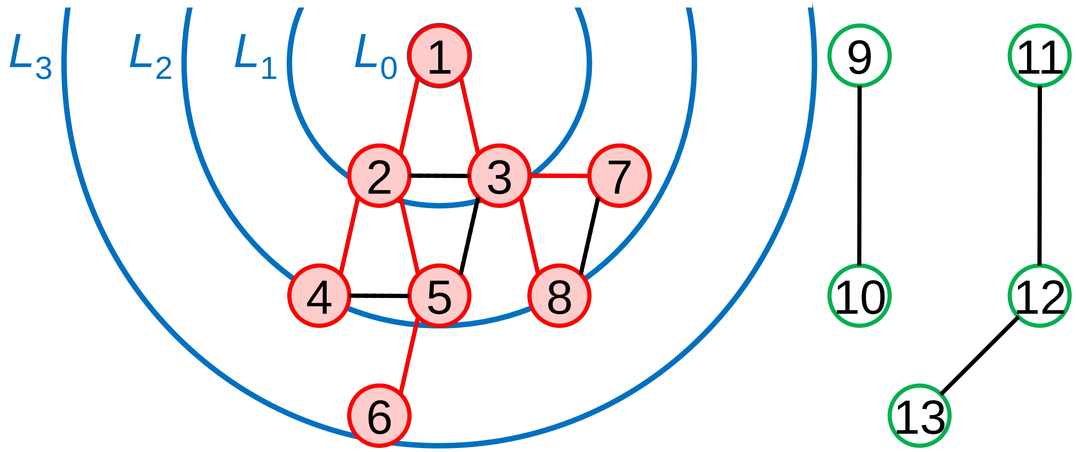
Node-to-Node Connectivity

- Q: Given a graph $G = (V, E)$ and two particular nodes s and t, is there a path from s to t in G?
  - The s-t connectivity problem
  - The maze-solving problem
- A:
  - For small graphs, easy! (visual inspection)
    - 1-6 connectivity? 7-13 connectivity?
  - What if large graphs? How efficiently can we do?



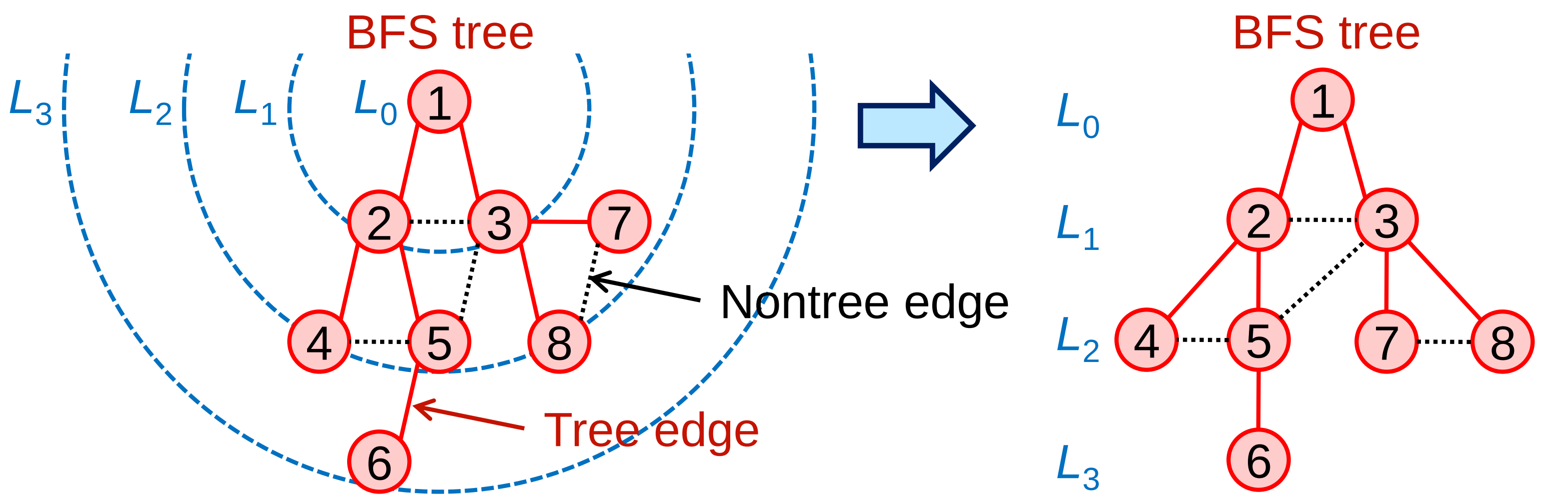
Breadth-First-Search (BFS)

- Breadth-first search (BFS): propagate the waves
  - Start at s and flood the graph with an expanding wave that grows to visit all nodes that it can reach.
  - Layer L_i: i is the time that a node is reached.
    - Layer L_0 = {s}; layer L_1 = all neighbors of L_0. Adjacent nodes
    - Layer L_{j+1} = all nodes that do not belong to an earlier layer and that are neighbors of L_j.
    - i = distance between s to the nodes that belong to layer L_i.



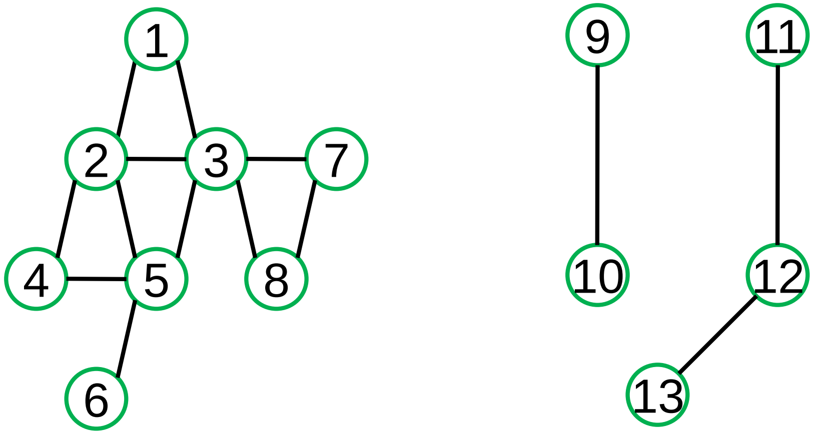
BFS Tree

- Let T be a BFS tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge of G. Then i and j differ by at most 1.
- Pf:
  - Without loss of generality, suppose $j - i > 1$.
  - By definition, $x \in L_i$, x's neighbors belongs to L_{i+1} or earlier.
  - Since (x, y) is an edge of G, y is x's neighbor, $y \in L_j$ and $j \leq i+1$.
  - $\rightarrow\leftarrow$



Connected Component

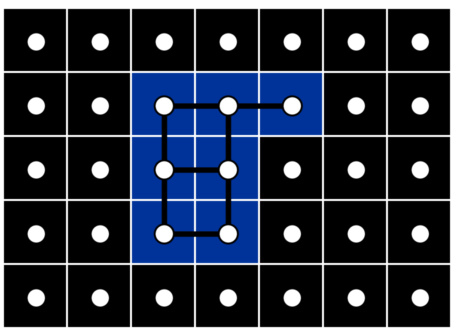
- A connected component containing *s* is the set of nodes that are reachable from *s*.
  - Connected component containing node 1 is {1, 2, 3, 4, 5, 6, 7, 8}.
  - There are three connected components.
    - The other two are {9, 10} and {11, 12, 13}.



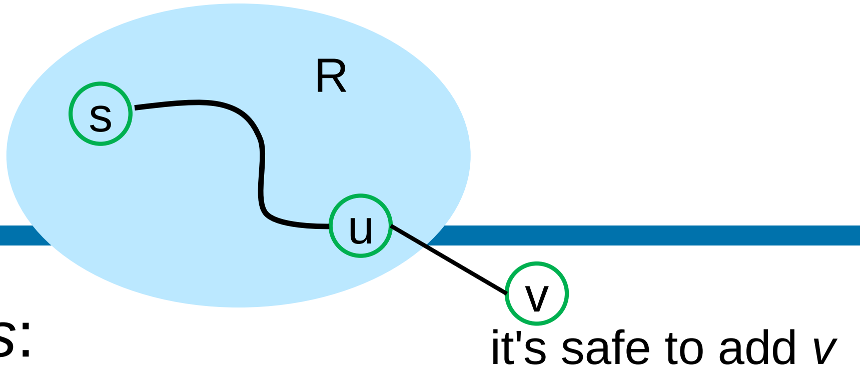
Color Fill

- Q: Given lime green pixel in an image, how to change color of entire blob of neighboring lime pixels to blue?
- A: Model the image as a graph.
  - Node: pixel.
  - Edge: two neighboring lime pixels.
  - Blob: connected component of lime pixels.

recolor lime green blob to blue



Connected Component



- Find all nodes reachable from *s*:

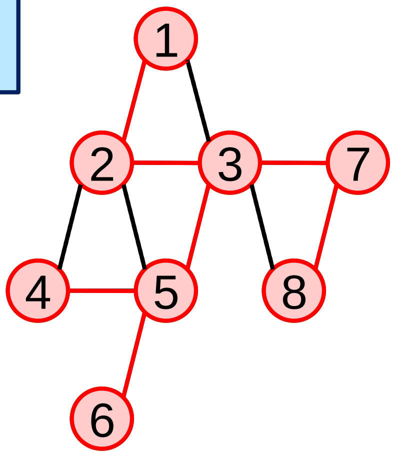
> Connected-Component(*s*)
> // *R* will consist of nodes to which *s* has a path
> 1. initialize $R = \{s\}$
> 2. **while** (there is an edge (u, v) where $u \in R$ and $v \notin R$) **do**
> 3. $\quad R = R + \{v\}$

- Correctness: Upon termination, *R* is the connected component containing *s*.
- Pf:
  - Q: How about any node $v \in R$?
  - Q: How about a node $w \notin R$?
- Q: How to recover the actual path from *s* to any node $t \in R$?
- Q: How to explore a new edge in line 2?
  - BFS: explore in order of distance from *s*.
  - Any method else?

Depth-First Search (DFS)

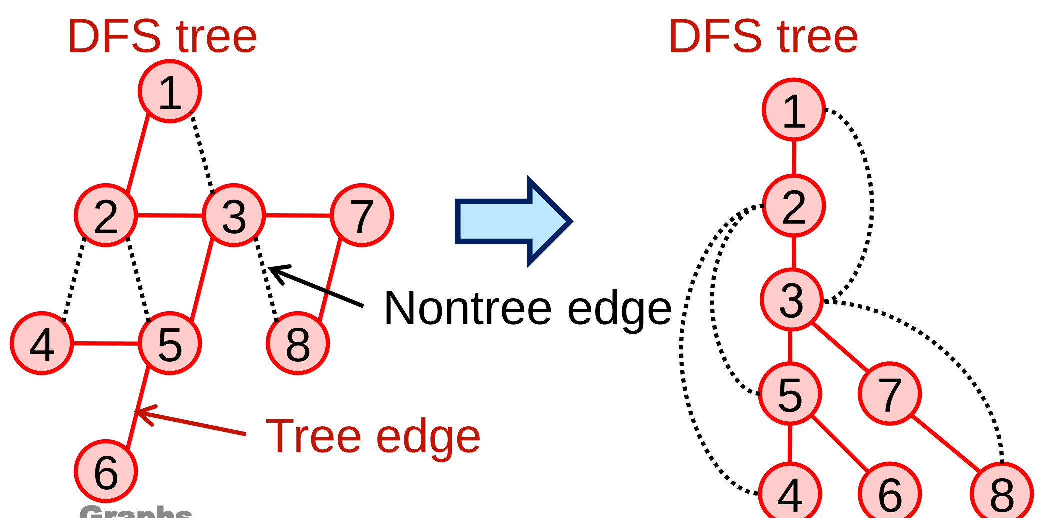
- Depth-first search (DFS): Go as deeply as possible or retreat
  - Start from *s* and try the first edge leading out, and so on, until reach a dead end. Backtrack and repeat.
    - A mouse in a maze without the map.
  - Another method for finding connected component

DFS(*u*)
1. mark *u* as explored and add *u* to *R*
2. **foreach** edge (*u*, *v*) incident to *u* **do**
3.   **if** (*v* is not marked as explored) **then**
4.   recursively invoke DFS(*v*)



DFS Tree

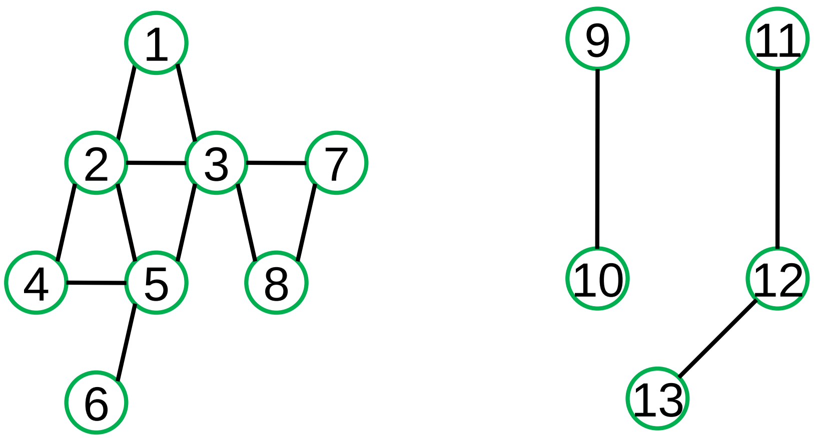
- Let T be a DFS tree, x and y nodes in T, and (x, y) a nontree edge. Then one of x or y is an ancestor of the other.
  descendant
- Pf:
  - WLOG, suppose x is reached first by DFS.
  - When (x, y) is examined during DFS(x), it is not added to T because y is marked explored.
  - Since y is not marked as explored when DFS(x) was first invoked, it is a node that was discovered between the invocation and end of the recursive call DFS(x).
  - y is a descendant of x.



Summary: BFS and DFS

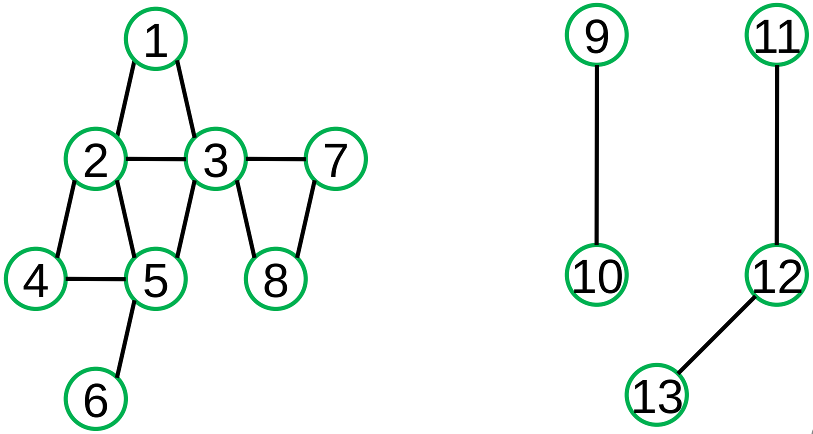
- Similarity: BFS/DFS builds the connected component containing *s*.
- Difference: BFS tree is flat/short; DFS tree is narrow/deep.
  - What are the nontree edges in BFS/DFS?

- Q: How to produce all connected components of a graph?
- A:



Implementation

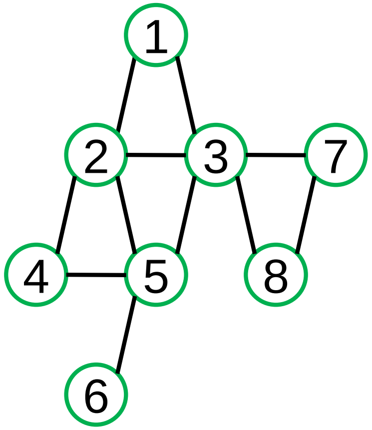
Lists / arrays
Queues / stacks



Representing Graphs

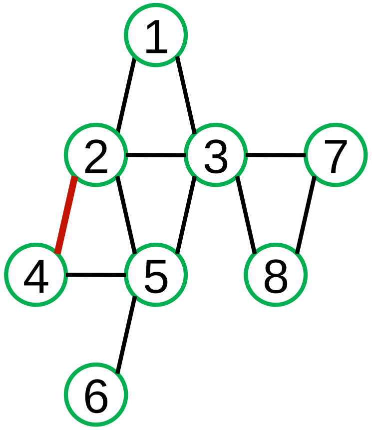
- A graph $G = (V, E)$
  - $|V|$ = the number of nodes = n
  - $|E|$ = the number of edges = m

    cardinality (size) of a set
- Dense or sparse?
  - For a connected graph, $n - 1 \le m \le \binom{n}{2} \le n^2$
- Linear time = $O(m+n)$
  - Why? It takes $O(m+n)$ to read the input



Adjacency Matrix

- Consider a graph $G = (V, E)$ with n nodes, $V = \{1, \ldots, n\}$.
- The adjacency matrix of G is an nxn martix A where
  - $A[u, v] = 1$ if $(u, v) \in E$;
  - $A[u, v] = 0$, otherwise.
- Time:
  - $\Theta(1)$ time for checking if $(u, v) \in E$.
  - $\Theta(n)$ time for finding out all neighbors of some $u \in V$.
    - Visit many 0's
- Space: $\Theta(n^2)$
  - What if sparse graphs?

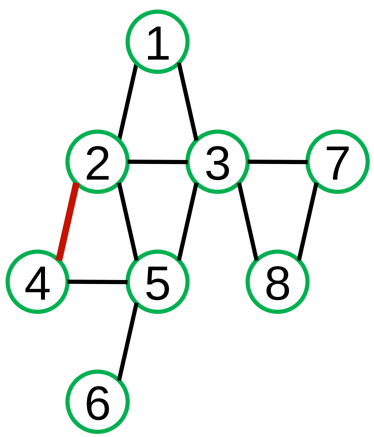


symmetric

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Adjacency List

- The adjacency list of *G* is an array *Adj*[]of *n* lists, one for each node represents its neighbors
  - *Adj*[*u*] = a linked list of $\{v \mid (u, v) \in E\}$.
- Time: degree of *u*: number of neighbors
  - $\Theta(\deg(u))$ time for checking one edge or all neighbors of a node.
- Space: $O(n+m)$



What is a Queue?

Two open ended container

- A queue is a set of elements from which we extract elements in first-in, first-out (FIFO) order.
  - We select elements in the same order in which they were added.

$Q = (a_1, ..., a_n)$

front rear

front rear

Pop a_1 a_2 … a_n Push

Push (A) A

front, rear

Push (B) A B

front rear

Push (C) A B C

front rear

Push (D) A B C D

front rear

Pop () B C D

front rear

Push (E) B C D E

front rear

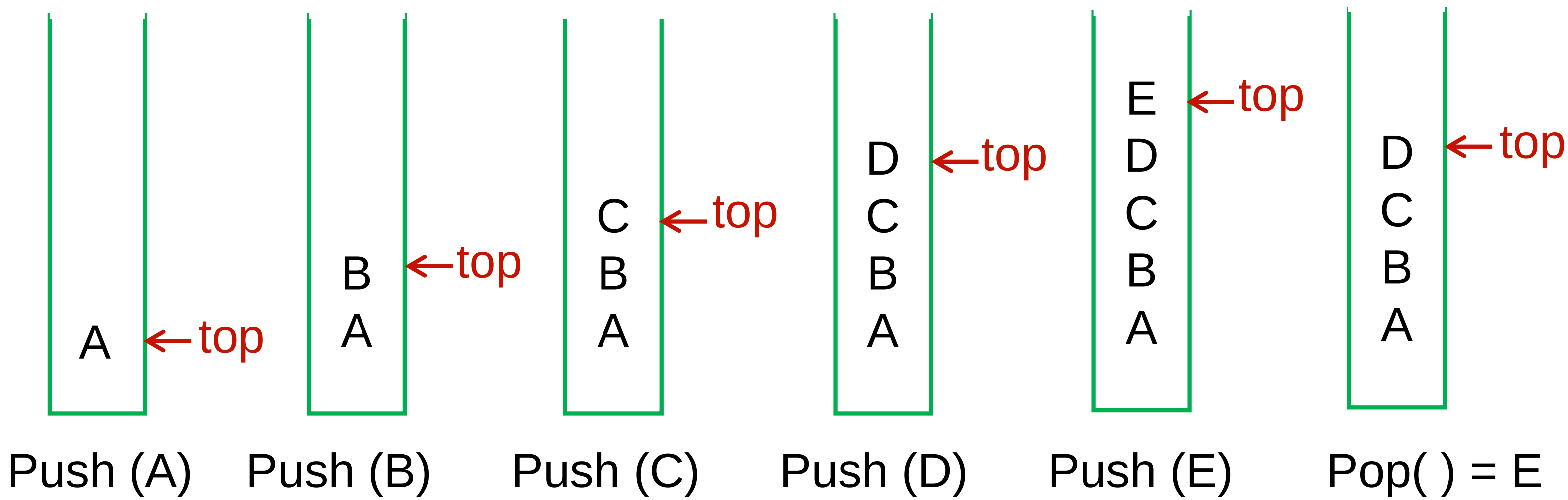
What is a Stack?

One open ended and one close ended container

- A stack is a set of elements from which we extract elements in last-in, first-out (LIFO) order.
  - Each time we select an element, we choose the one that was added most recently.

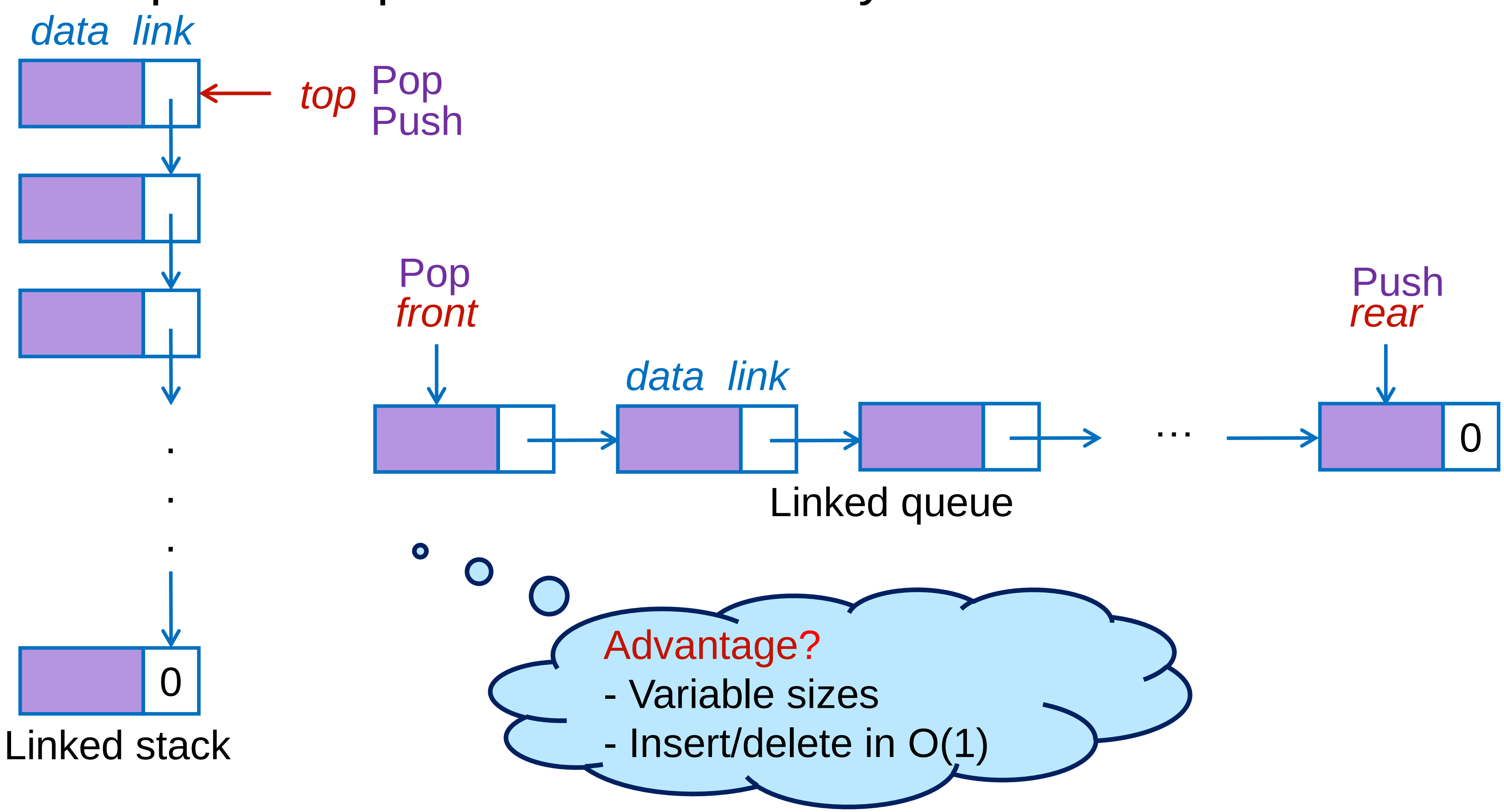
$$S = (a_1, \ldots, a_n)$$

bottom: a_1 — top: a_n



Linked Stacks and Queues

- Implement queues and stacks by linked lists

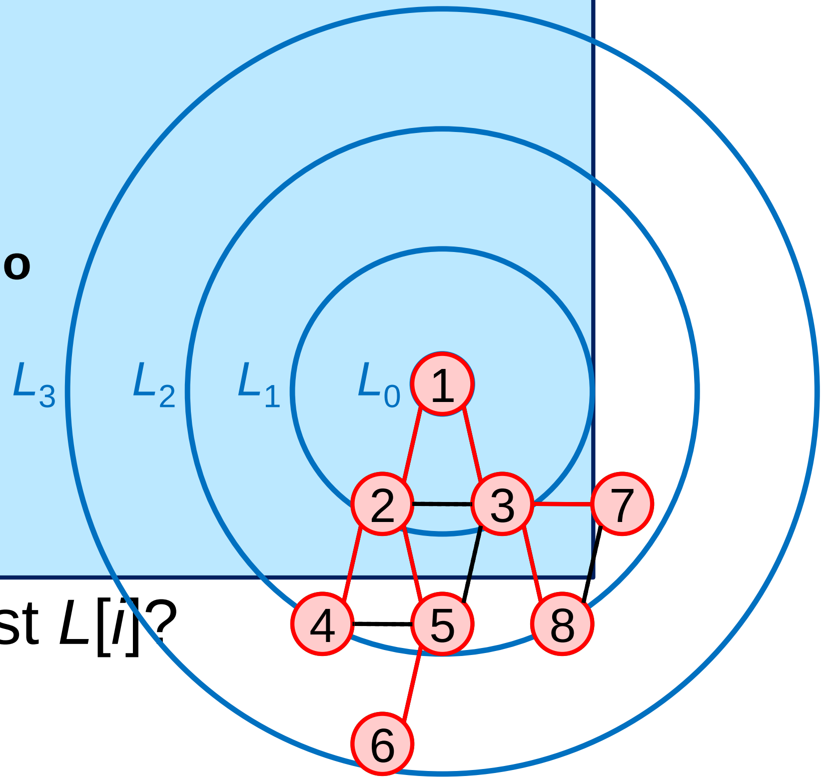


Implementing BFS

Q: Running time?
$O(n^2)$ or $O(n+m)$
Why?

- Adjacency list is ideal for implementing BFS

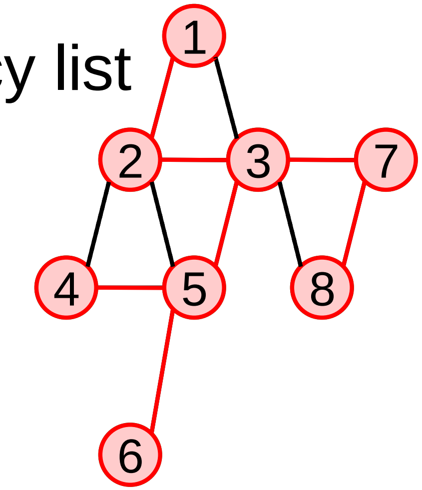
```
BFS(s) // T will be BFS tree rooted at s; layer counter i; layer list L[i]
Discovered[s] = true; Discovered[v] = false for other v
i = 0; L[0] = {s}; T = {};
while (L[i] is not empty) do
    L[i+1] = {};
    for each (node u∈L[i]) do
        for each (edge (u, v) incident to u) do
            if (Discovered[v] = false) then
                Discovered[v] = true
                T = T + {(u, v)}
                L[i+1] = L[i+1] + {v}
    i++
```



- Q: How to manage each layer list $L[i]$?
- A: A queue/stack is fine
  - Nodes in $L[i]$ can be in any order
  - Or, we can merge all layer lists into a single list L as a queue

Implementing DFS

- We implement DFS based on adjacency list
- Recursive procedure




```
DFS(u)
mark u as explored and add u to R
foreach edge (u, v) incident to u do
    if (v is not marked as explored) then
        recursively invoke DFS(v)
```

- Alternative implementation of DFS

```
DFS(s)
// S: a stack of nodes whose neighbors haven't been entirely explored
S = {s}
while (S is not empty) do
    remove a node u from S
    if (Explored[u] = false) then
        Explored[u] = true
        for each (edge (u, v) incident to u) do
            S = S + {v}
```

Q: Running time?
O(n^2) or O($n+m$)
Why?

Summary: Implementation

- Graph:

Adjacency matrix vs. Adjacency list	Winner
Faster to find an edge?	Matrix
Faster to find degree?	List
Faster to traverse the graph?	List
Storage for sparse graph?	List
Storage for dense graph?	Matrix
Edge insertion or deletion?	Matrix
Better for most applications?	List

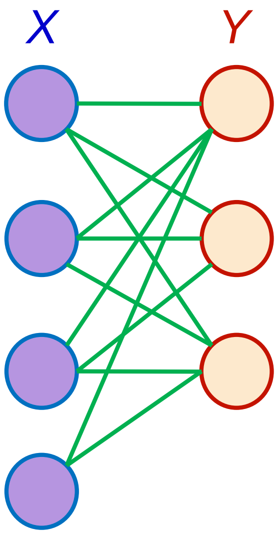
- Graph traversal
  - BFS: queue (or stack)
  - DFS: stack
  - O(n+m) time

Testing Bipartiteness

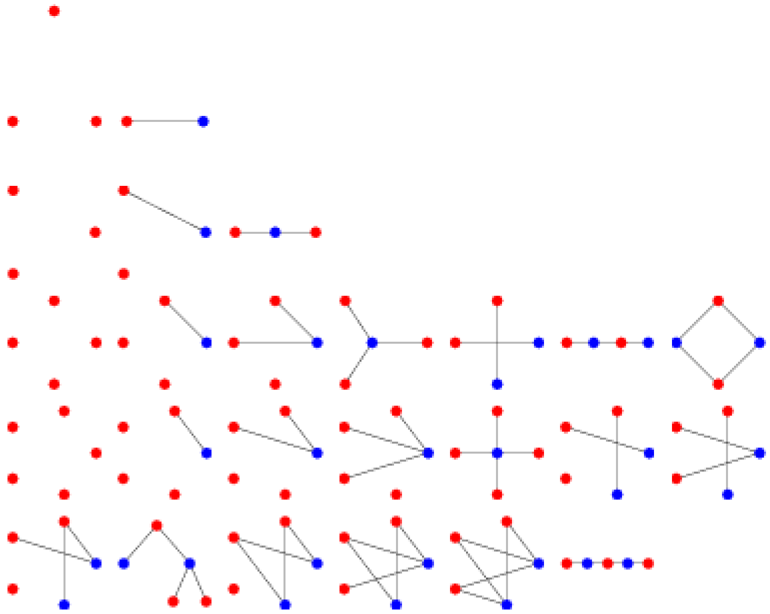
Application of BFS

Bipartite Graphs

- A bipartite graph (bigraph) is a graph whose nodes can be partitioned into sets *X* and *Y* in such a way that every edge has one one end in *X* and the other end in *Y*.
  - *X* and *Y* are two disjoint sets.
  - No two nodes within the same set are adjacent.



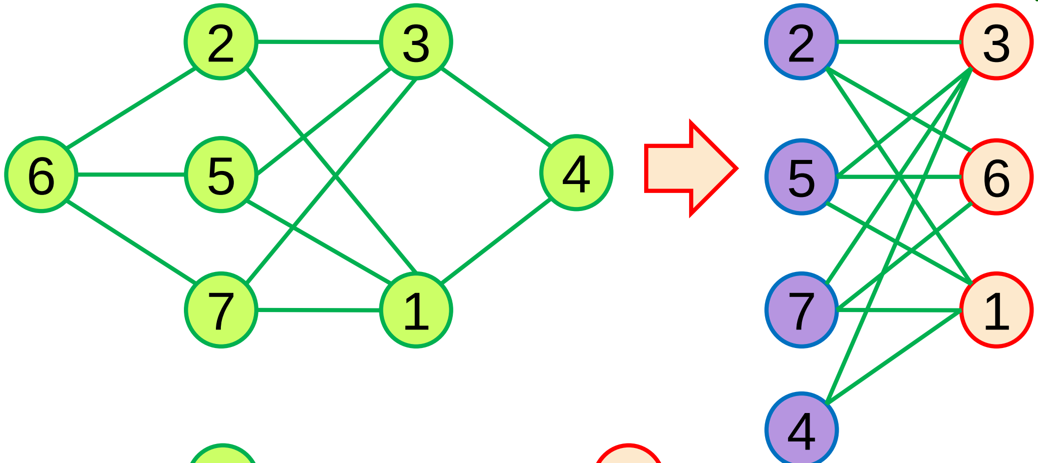
$G = (X, Y, E)$

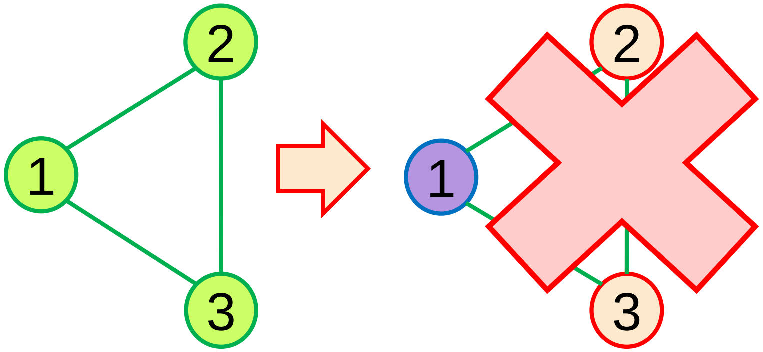


http://mathworld.wolfram.com/BipartiteGraph.html

Is a Graph Bipartite?

- Q: Given a graph *G*, is it bipartite?
- A: Color the nodes with blue and red (two-coloring)

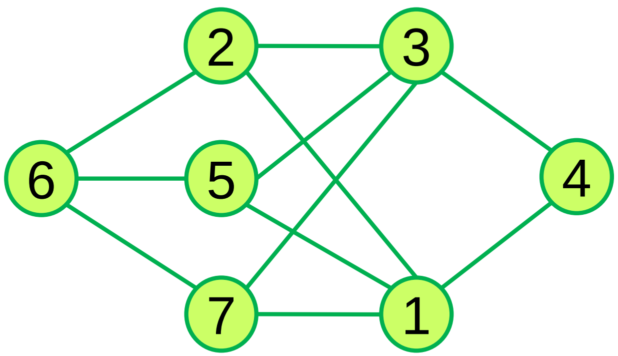


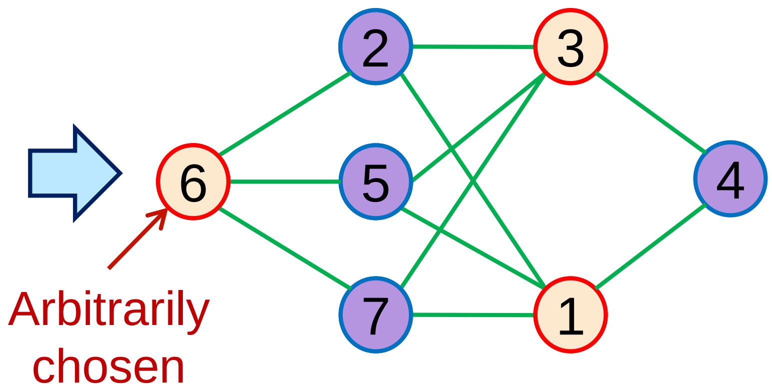


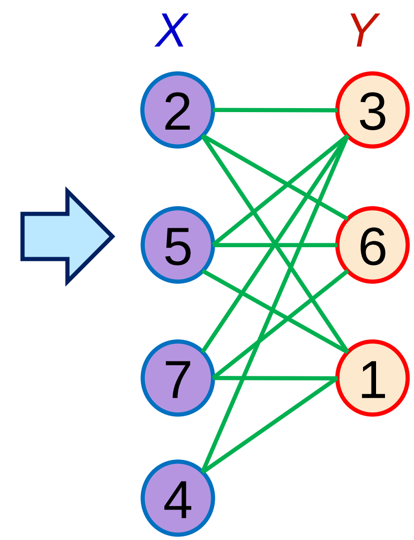
- If a graph *G* is bipartite, then it cannot contain an odd cycle.

Testing Bipartiteness

- Color the nodes with blue and red



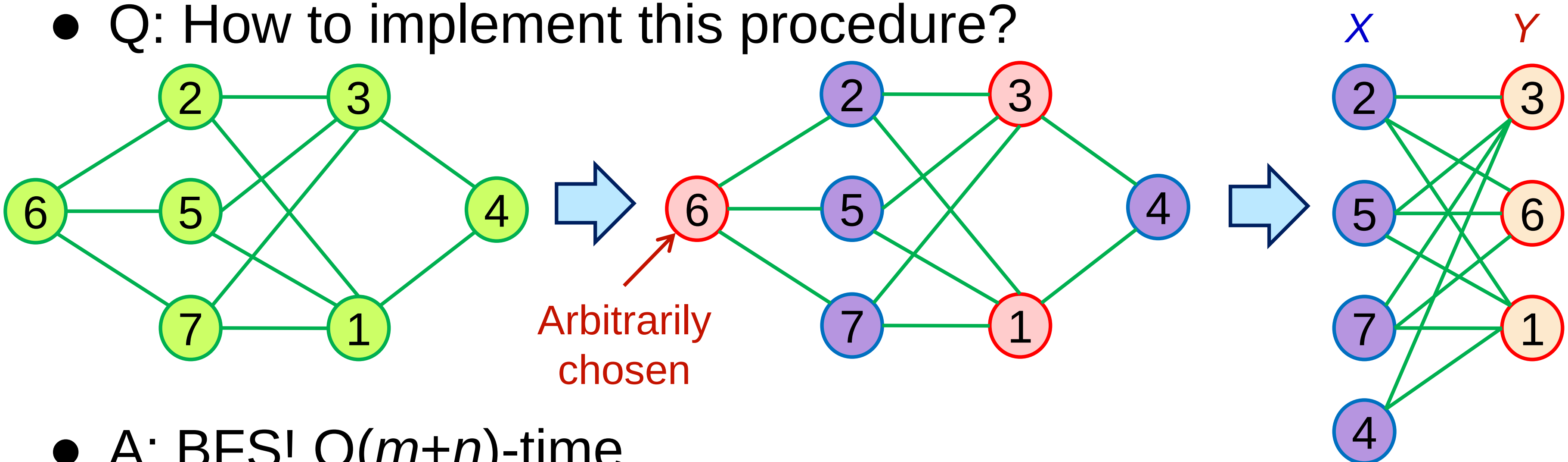




- Procedure:
  1. Assume $G = (V, E)$ is connected.
     - Otherwise, we analyze connected components separately.
  2. Pick any node $s \in V$ and color it red
     - Anyway, s must receive some color.
  3. Color all the neighbors of s blue.
  4. Repeat coloring red/blue until the whole graph is colored.
  5. Test bipartiteness: every edge has ends of opposite colors.

Implementation: Testing Bipartiteness

- Q: How to implement this procedure?

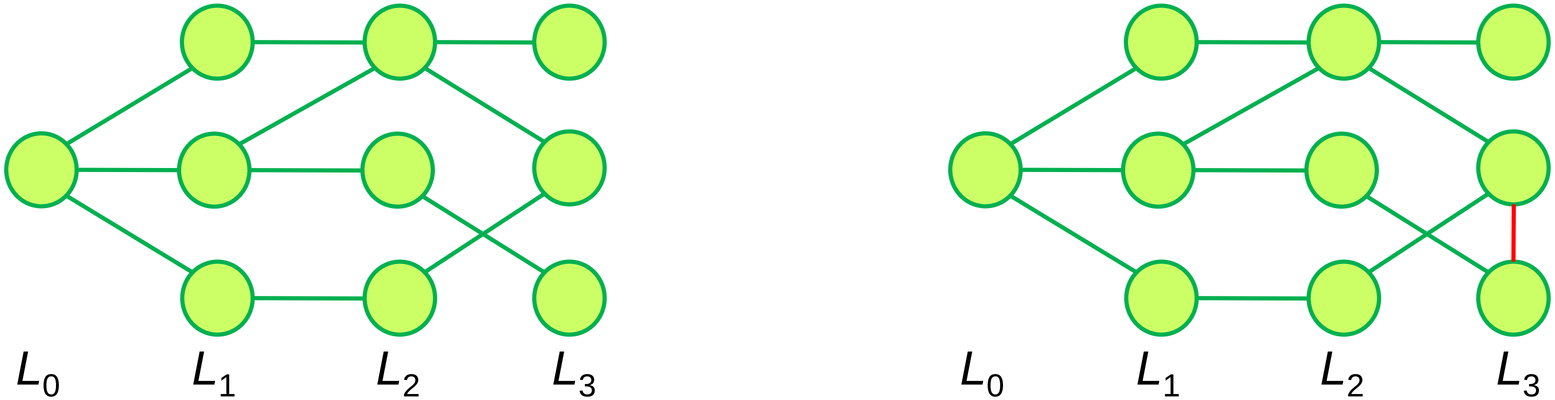


- A: BFS! O(m+n)-time
  - We perform BFS from any s, coloring s red, all of layer L_1 blue, …
    - Even/odd-numbered layers red/blue
    - Insert the following statements after line 10 of BFS(s) (p. 34)

10.	$L[i+1] = L[i+1] + \{v\}$
10a.	**if** ((i+1) is even) **then**
10b.	Color[v] = red
10c.	**else**
10d.	Color[v] = blue

Proof: Correctness (1/2)

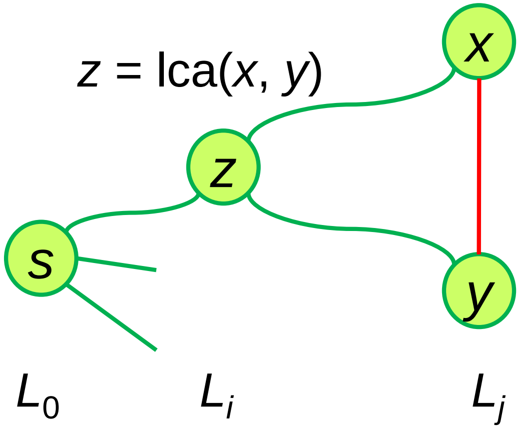
- Let G be a connected graph and let L_0, L_1, … be the layers produced by BFS starting at node s. Then exactly one of the following holds.
  1. No edge of G joins two nodes of the same layer, and G is bipartite.
  2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).
- Pf: Case 1 is trivial.



Let $x \in L_i$, $y \in L_j$, and $(x, y) \in E$.
Then i and j differ by at most 1.

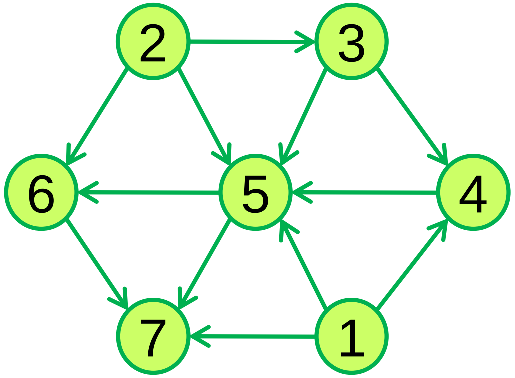
Proof: Correctness (2/2)

- Pf: (Case 2)
  - Suppose (x, y) is an edge with x, y in same layer L_j.
  - Let $z = \text{lca}(x, y)$ = lowest common ancestor.
  - Let L_i be the layer containing z.
  - Consider the cycle that takes edge from x to y, then path from y to z, then path from z to x.
  - Its length is $\underbrace{1}_{(x,\, y)} + \underbrace{(j-i)}_{y \text{->} z} + \underbrace{(j-i)}_{z \text{->} x}$, which is odd.



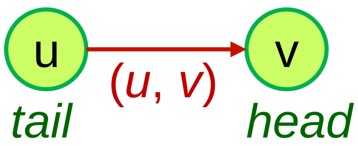
Let $x \in L_i$, $y \in L_j$, and $(x, y) \in E$.
Then i and j differ by at most 1.

Connectivity in Directed Graphs

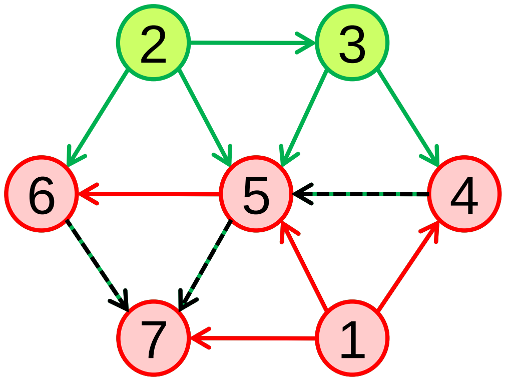


Recap: Directed Graphs

- In a directed graph: asymmetric relationships
  - Edges are directed, i.e., (u, v) != (v, u)
    - e.g., u knows v (celebrity), while v doesn't know u.
    - Directionality is crucial.

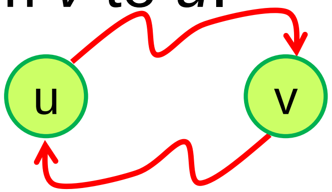


- Representation: Adjacency list
  - Each node is associated with two lists, instead of one in an undirected graph.
    - To which
    - From which
- Graph search algorithms: BFS/DFS
  - Almost the same as undirected graphs
    - Again, directionality is crucial.
    - Q: What can we reach from node 1?
    - A:



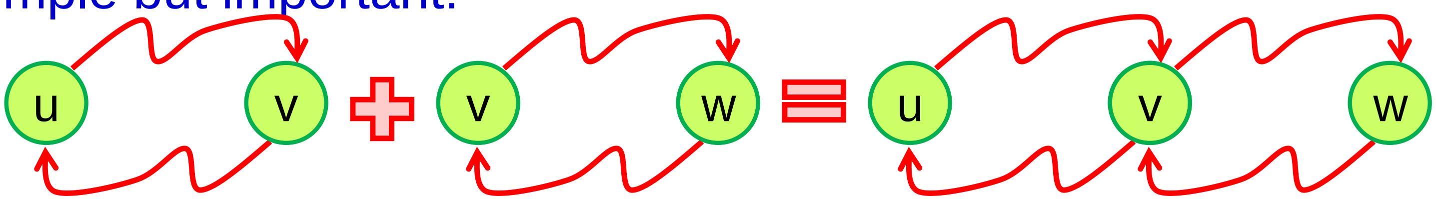
Strong Connectivity

- Nodes *u* and *v* are mutually reachable if there is a path from *u* to *v* and also a path from *v* to *u*.



- A directed graph is strongly connected if every pair of nodes are mutually reachable.
  - Q: What kind of graph has no mutually reachable nodes?

- Lemma: If *u* and *v* are mutually reachable, and *v* and *w* are mutually reachable, then *u* and *w* are mutually reachable.
  - Simple but important!
- Pf:



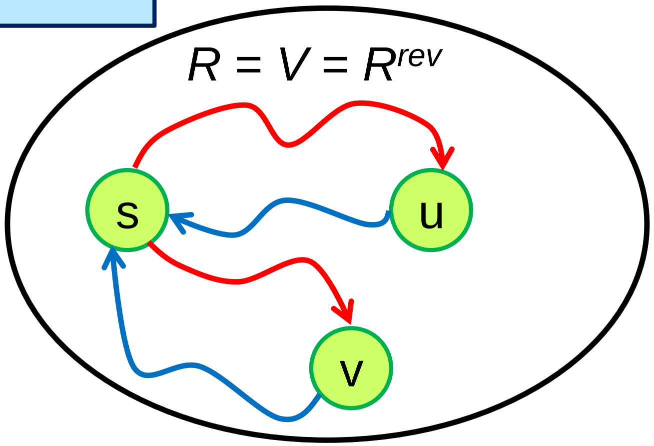
Testing Strong Connectivity

- Q: Can we determine if a graph is strongly connected in linear time?
- A: Yes. How? Why?

TestSC(G)
1. pick any node s in G
2. R = BFS(s, G)
3. R^{rev} = BFS(s, G^{rev})
4. **if** ($R = V = R^{rev}$) **then return** true **else** false

G^{rev}: reverse the direction of every edge in G
(u, v) in G^{rev} if (v, u) in G

  - Time: O(m+n)
  - Q: Correctness?



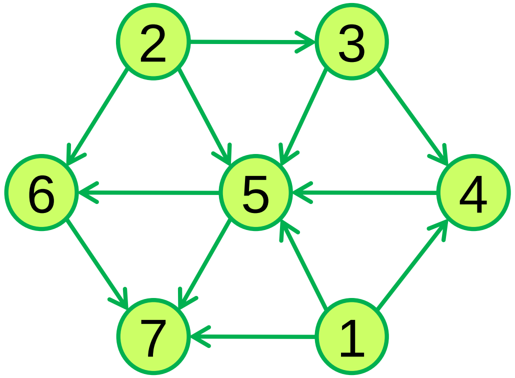
  - Q: How to partition a directed graph into strong components?

Strong Component

- The strong component containing *s* in a directed graph is the maximal set of all *v* s.t. *s* and *v* are mutually reachable.
  - a.k.a. strongly connected component

- Theorem: For any two nodes *s* and *t* in a directed graph, their strong components are either identical or disjoint.
  - Q: When are they identical? When are they disjoint?
- Pf:
  - Identical if *s* and *t* are mutually reachable
    - *s* -- *v*, *s* -- *t*, *v* -- *t*
  - Disjoint if *s* and *t* are not mutually reachable
    - Proof by contradiction

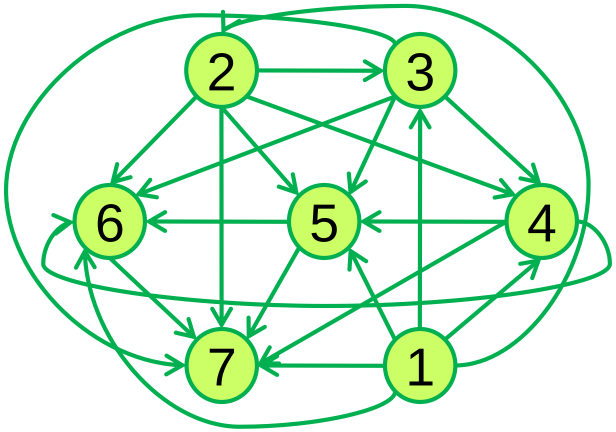
DAGs and Topological Ordering



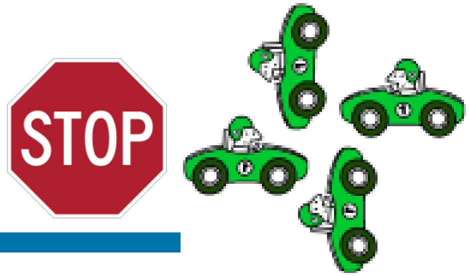
Directed Acyclic Graphs

- Q: If an undirected graph has no cycles, then what's it?
- A: A tree (or forest).
  - At most *n*-1 edges.

- A directed acyclic graph (DAG) is a directed graph without cycles.
  - A DAG may have a rich structure.
  - A DAG encodes dependency or precedence constraints
    - e.g., prerequisite of Algorithms:
      Data structures
      Discrete math
      Programming C/C++
    - e.g., execution order of instructions in CPU
      Pipeline structures



Topological Ordering

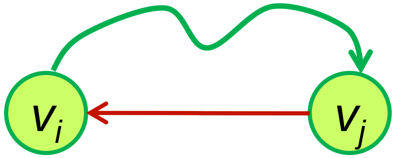


- Q: 4 drivers come to the junction simultaneously, who goes first?
  - Deadlock! Dependencies form a cycle!

The driver must come to a complete stop at a stop sign. Generally the driver who arrives and stops first continues first. If two or three drivers in different directions stop simultaneously at a junction controlled by stop signs, generally the drivers on the left must yield the right-of-way to the driver on the far right.

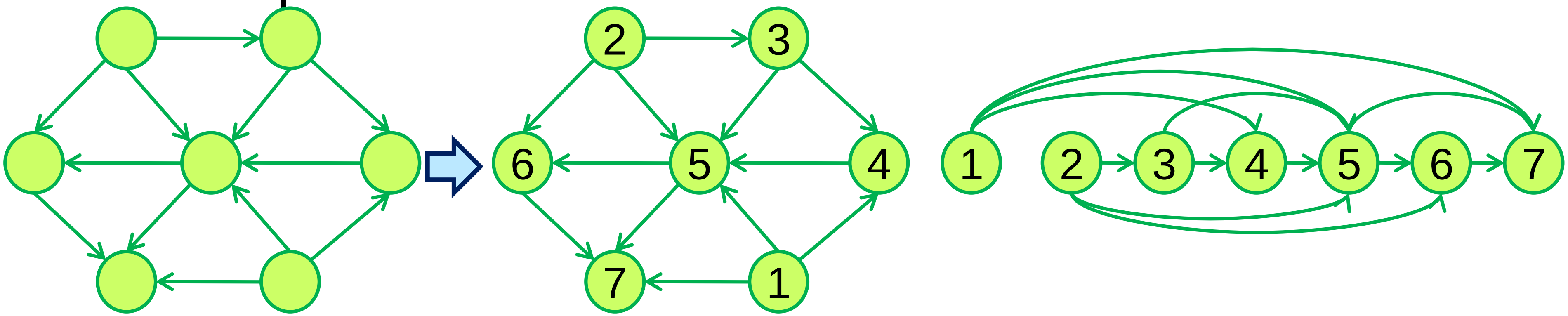
- Given a directed graph G, a topological ordering is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j), we have $i<j$.
  - Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.

- Lemma: If G has a topological ordering, then G is a DAG.
- Pf: Proof by contradiction!
  - How? Consider a cycle, $v_i, \ldots, v_j, v_i$.



Example

- Example:



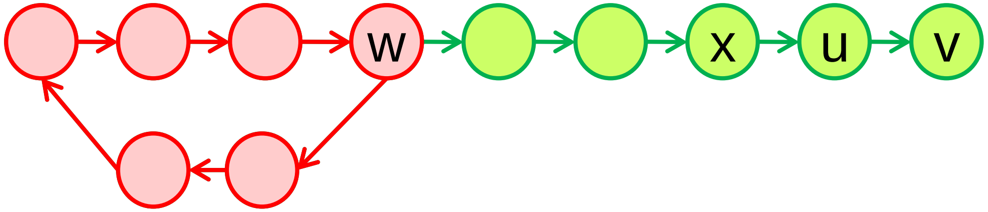
a DAG

the same DAG with topological ordering

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

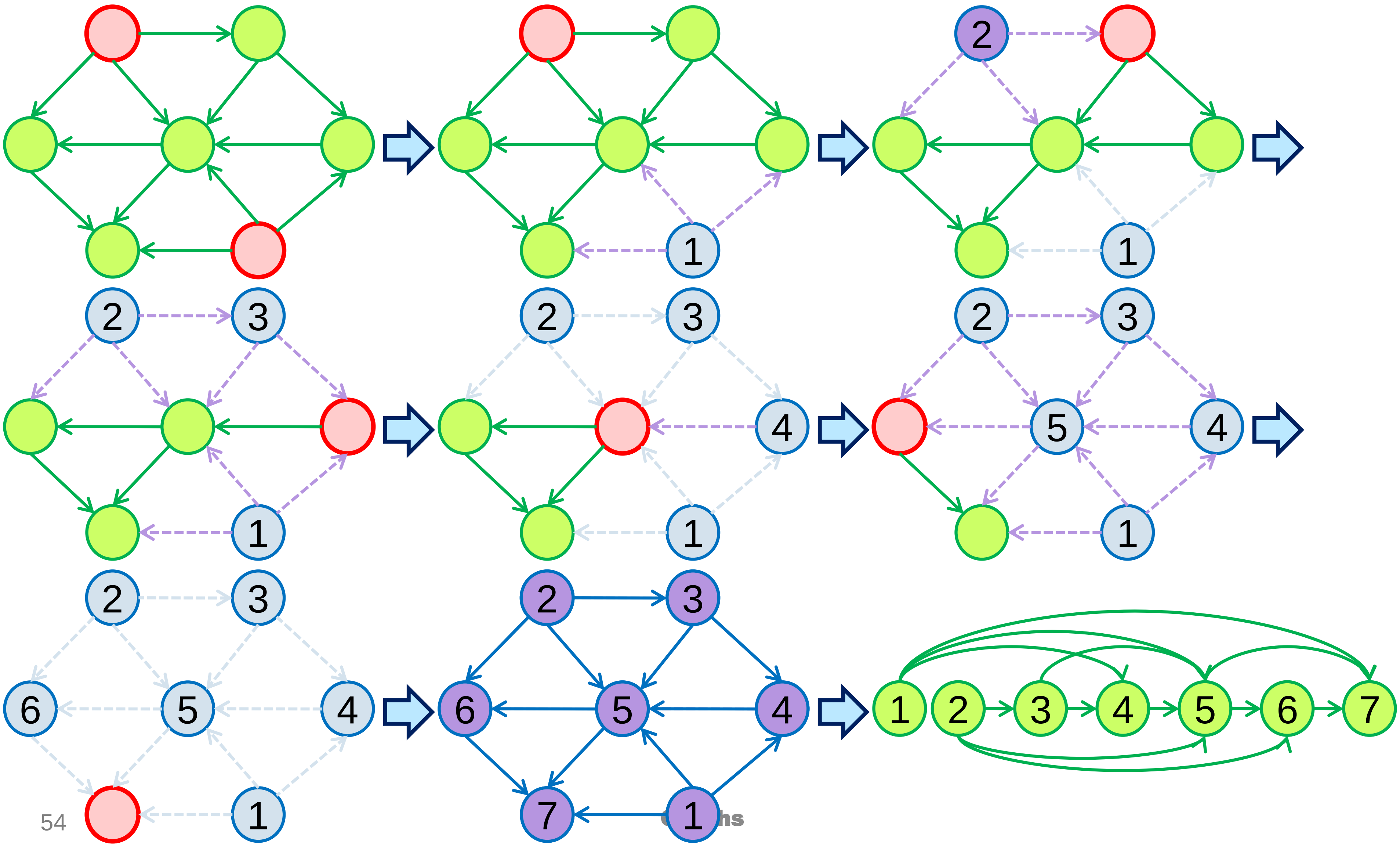
- Q: If *G* is a DAG, then does *G* have a topological ordering?
- Q: If so, how do we compute one?
- A: Key: find a way to get started!
  - Q: How?

Where to Start?



- A: A node that depends on no one, i.e., unconstrained.
- Lemma: In every DAG *G*, there is a node with no incoming edges.
- Pf: Proof by contradiction!
  - Suppose that *G* is a DAG where every node has at least one incoming edge. Let's see how to find a cycle in *G*.
  - 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*; and so on.
  - Repeat this process *n*+1 times (the initial *v* counts one). We will visit some node *w* twice, since *G* has only *n* nodes.
  - Let *C* denote the sequence of nodes encountered between successive visits to *w*. Clearly, *C* is a cycle. $\rightarrow\leftarrow$

Example: Topological Ordering

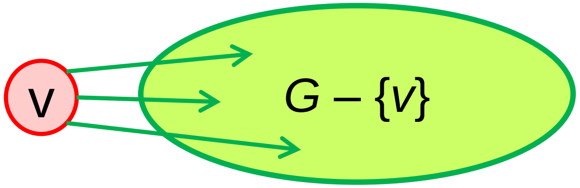


Topological Ordering

- Lemma: If G is a DAG, then G has a topological ordering.
- Pf: Proof by induction!

    1. Base case: true if $n = 1$.
    2. Inductive step:

    - Induction hypothesis: true for DAGs with up to n nodes
    - Given a DAG on n+1 nodes, find a node v w/o incoming edges.



$v \quad + \quad \langle v_1, v_2, \ldots, v_n \rangle \quad = \quad \langle v, v_1, v_2, \ldots, v_n \rangle$

    - $G - \{v\}$ is a DAG, since deleting v cannot create any cycles.
    - $G - \{v\}$ has n nodes. By induction hypothesis, $G - \{v\}$ has a topological ordering.
    - Place v first in topological ordering. This is safe since all edges of v point forward.
    - Then append nodes of $G - \{v\}$ in topological order after v.

A Linear-Time Algorithm

- In fact, the proof has already suggested an algorithm.

> TopologicalOrder(G)
> 1. find a node v without incoming edges
> 2. order v
> 3. $G = G - \{v\}$ // delete v from G
> 4. **if** (G is not empty) **then** TopologicalOrder(G)

- Time: From $O(n^2)$ to $O(m+n)$
  - $O(n^2)$-time: Total n iterations; line 1 in $O(n)$-time. How?
  - $O(m+n)$-time: How? Maintain the following information
    - indeg(w) = # of incoming edges from undeleted nodes
    - S = set of nodes without incoming edges from undeleted nodes
  - Initialization: $O(m+n)$ via single scan through graph
  - Update: line 3 deletes v
    - Remove v from S
    - Decrement indeg(w) for all edges from v to w, and add w to S if indeg(w) hits 0; this is O(1) per edge