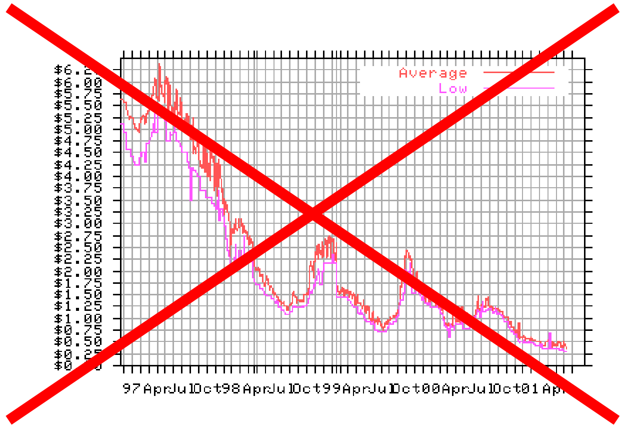
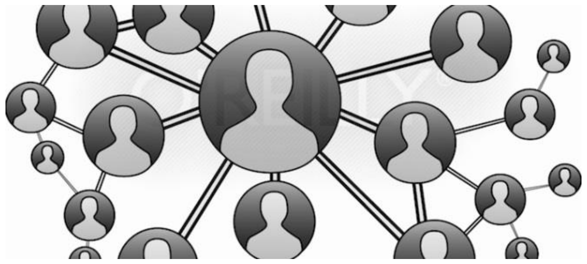
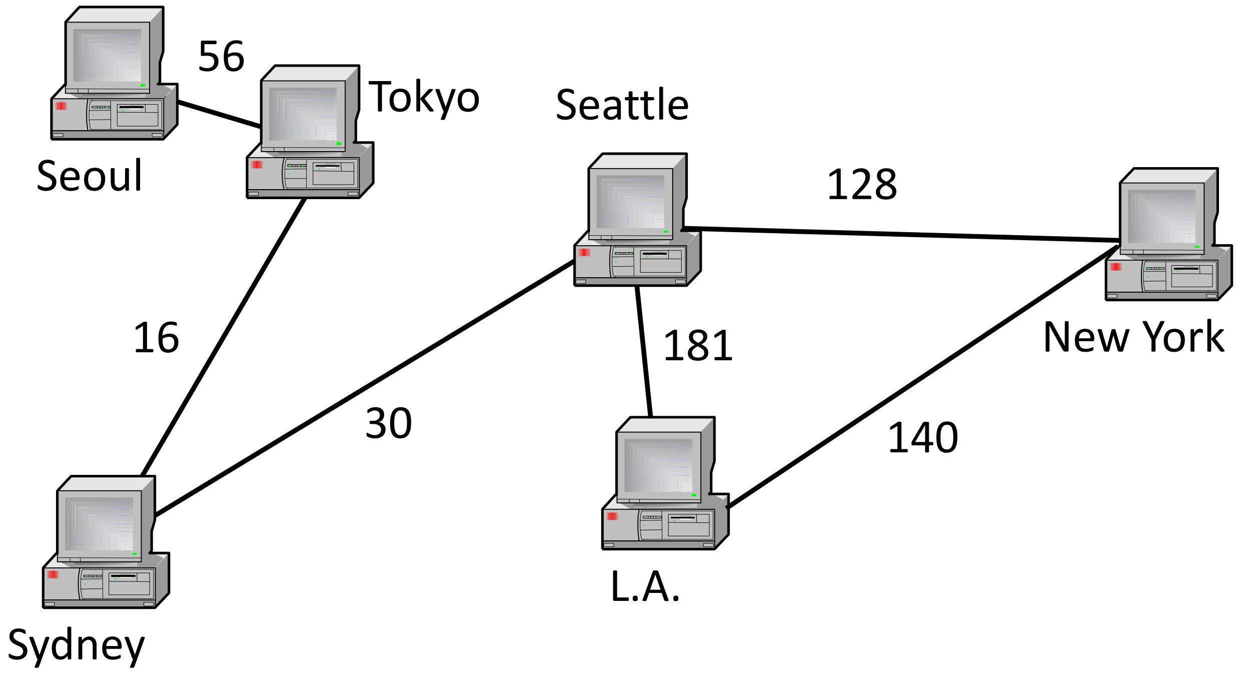
Graph Theory

Design and Analysis of Algorithms

What is a graph?

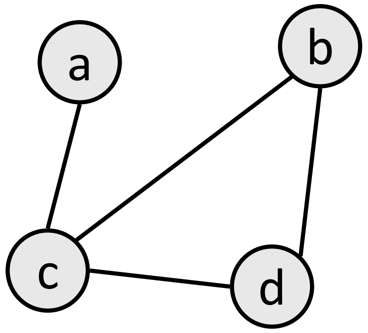






Graphs

- **graph**: A data structure containing:
  - a set of **vertices** V, *(sometimes called nodes)*
  - a set of **edges** E, where an edge represents a connection between 2 vertices.
    - Graph $G = (V, E)$
    - an edge is a pair (v, w) where v, w are in V

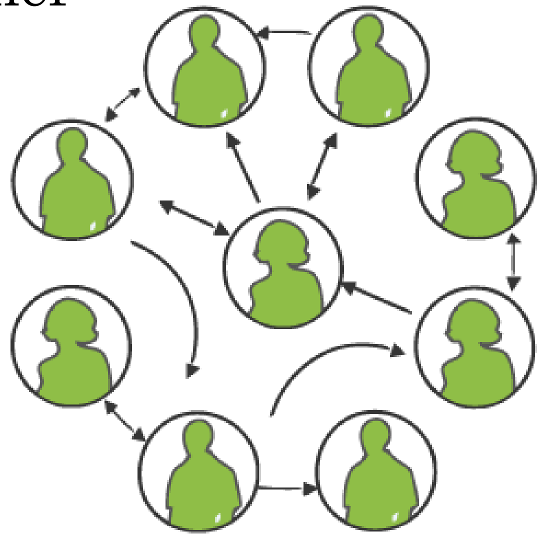


- the graph at right:
  - $V = \{a, b, c, d\}$
  - $E = \{(a, c), (b, c), (b, d), (c, d)\}$

- **degree**: number of edges touching a given vertex.
  - at right: a=1, b=2, c=3, d=2

Graph examples

- For each, what are the vertices and what are the edges?
  - Web pages with links
  - Network broadcast routing
  - Web crawling
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Facebook friends
  - Course pre-requisites
  - Family trees
  - Paths through a maze
  - Solving puzzles and games

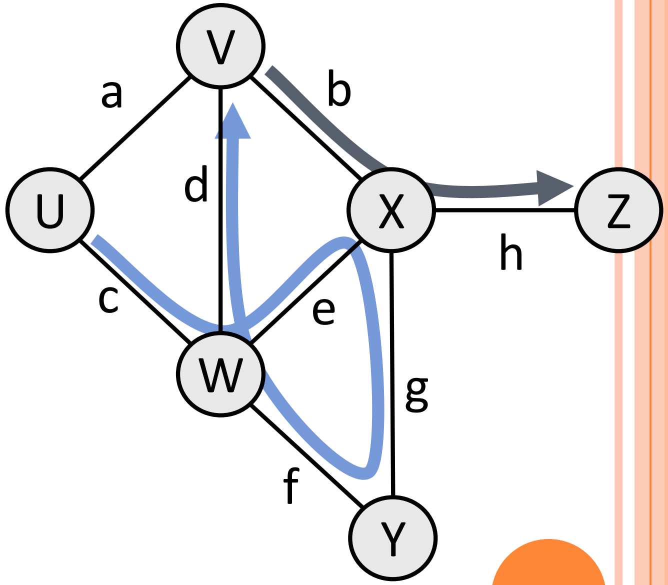


Applications of Graphs

- Driving Map
  - Edge = Road
  - Vertex = Intersection
  - Edge weight = Time required to cover the road
- Airline Traffic
  - Vertex = Cities serviced by the airline
  - Edge = Flight exists between two cities
  - Edge weight = Flight time or flight cost or both
- **Computer networks**
  - Vertex = Server nodes
  - Edge = Data link
  - Edge weight = Connection speed
- CAD/VLSI

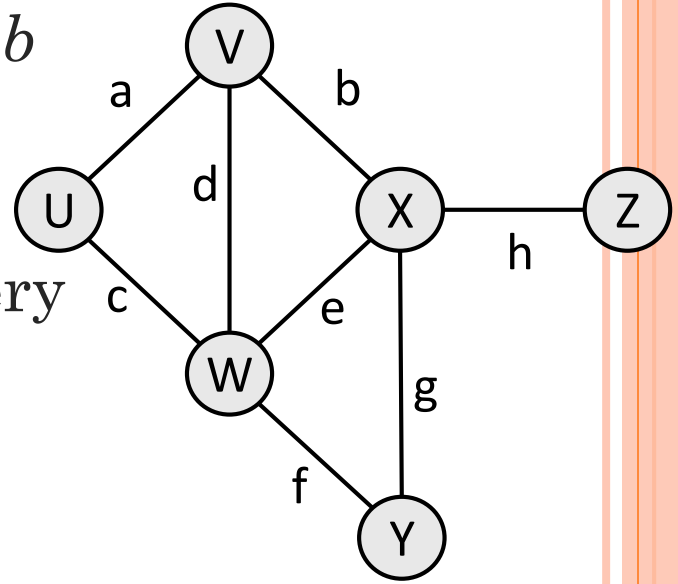
Paths

- **path**: A path from vertex *a* to *b* is a sequence of edges that can be followed starting from *a* to reach *b*.
  - can be represented as vertices visited, or edges taken
  - example, one path from *V* to *Z*: {b, h} or {V, X, Z}
  - What are two paths from U to Y?
- **path length**: Number of vertices or edges contained in the path.
- **neighbor** or **adjacent**: Two vertices connected directly by an edge.
  - example: V and X



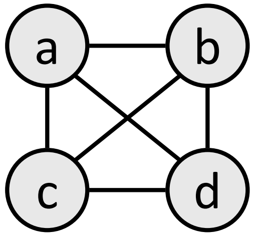
Reachability, connectedness

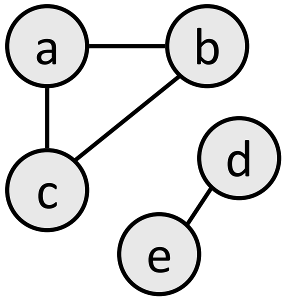
- **reachable**: Vertex *a* is *reachable* from *b* if a path exists from *a* to *b*.



- **connected**: A graph is *connected* if every vertex is reachable from any other.
  - Is the graph at top right connected?

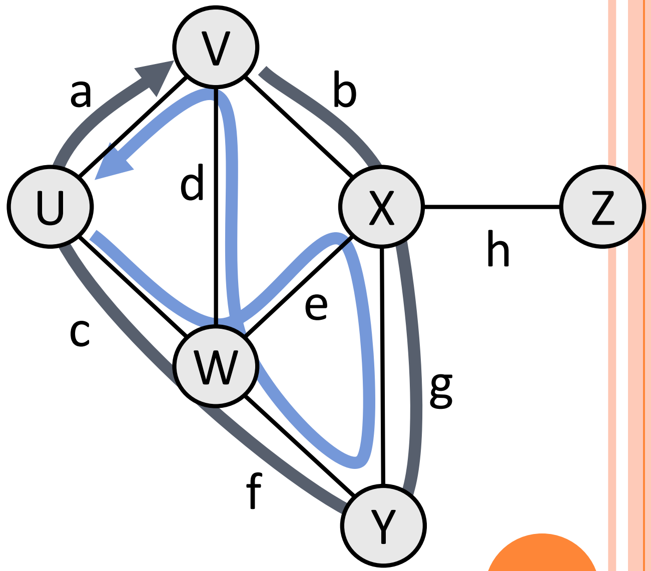
- **strongly connected**: When every vertex has an edge to every other vertex.





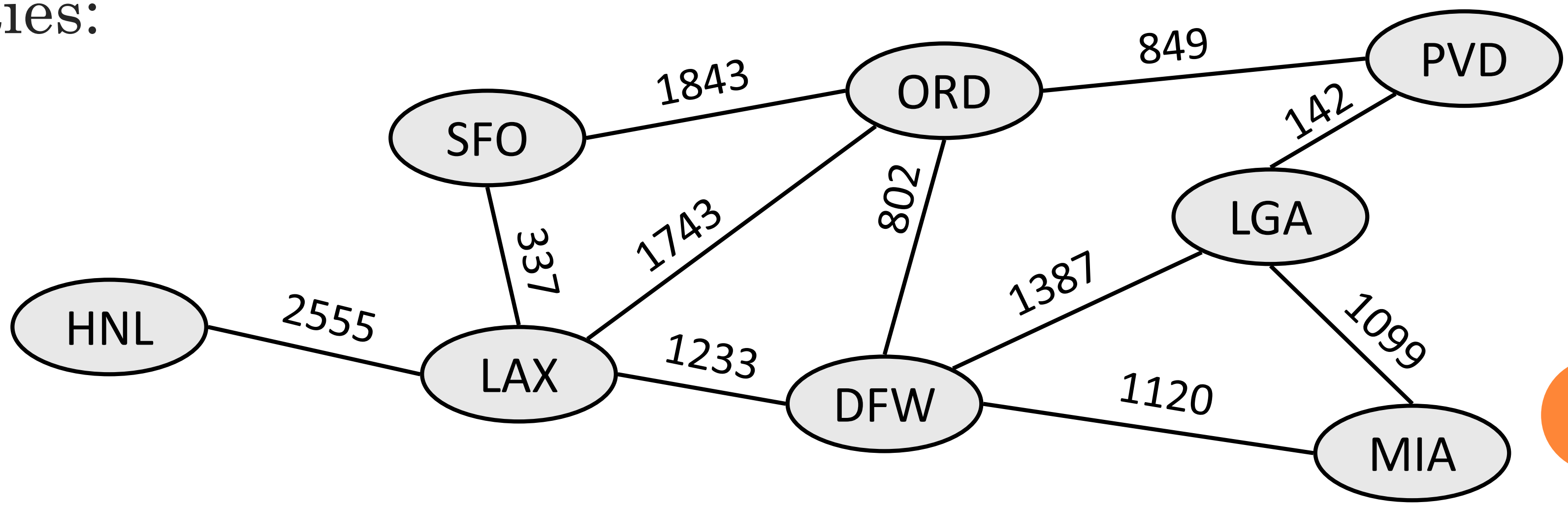
Loops and cycles

- **cycle**: A path that begins and ends at the same node.
  - example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
  - example: {c, d, a} or {U, W, V, U}.
  - **acyclic graph**: One that does not contain any cycles.
- **loop**: An edge directly from a node to itself.
  - Many graphs don't allow loops.



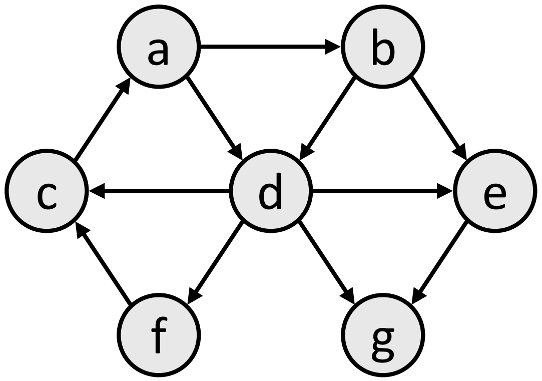
Weighted graphs

- **weight**: Cost associated with a given edge.
  - Some graphs have weighted edges, and some are unweighted.
  - Edges in an unweighted graph can be thought of as having equal weight (e.g. all 0, or all 1, etc.)
  - Most graphs do not allow negative weights.
- *example*: graph of airline flights, weighted by miles between cities:



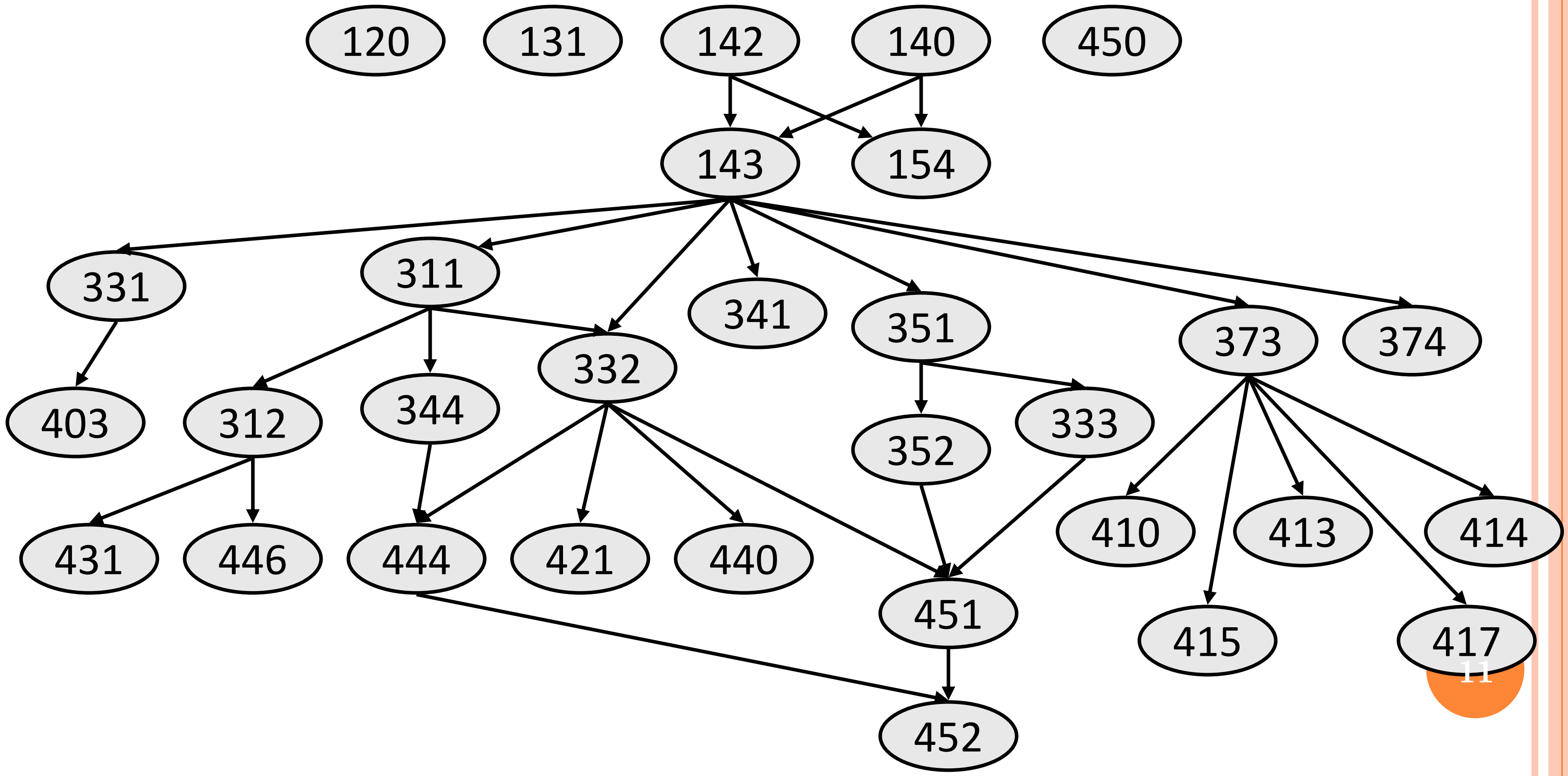
Directed graphs

- **directed graph** ("digraph"): One where edges are *one-way* connections between vertices.
  - If graph is directed, a vertex has a separate in/out degree.
  - A digraph can be weighted or unweighted.
  - Is the graph below connected? Why or why not?



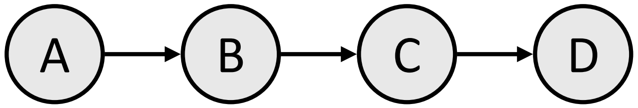
Digraph example

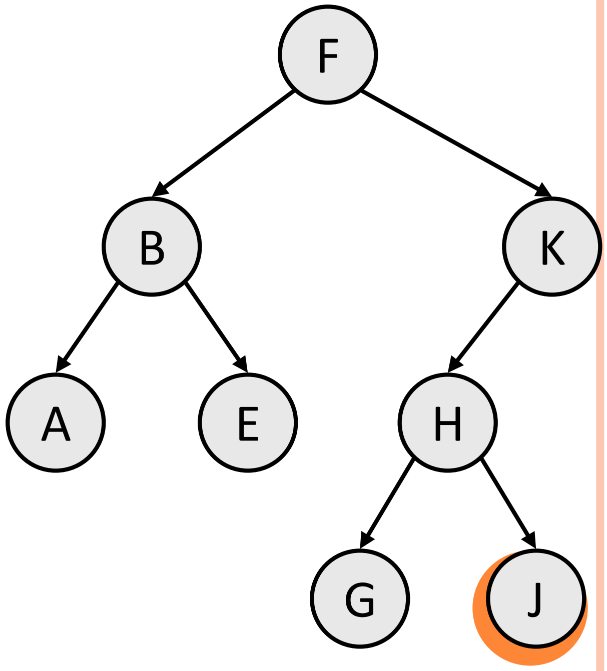
- Vertices= University courses (incomplete list)
- Edge (a, b) = *a* is a prerequisite for *b*



Linked Lists, Trees, Graphs

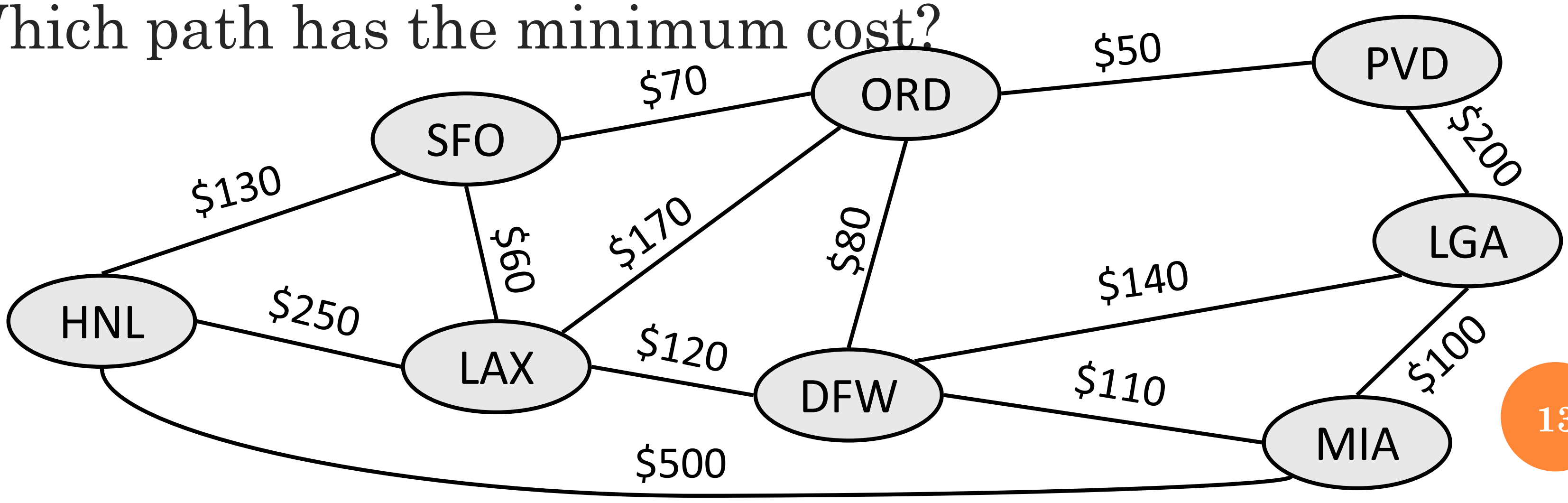
- **A *binary tree* is a graph with some restrictions:**
  - The tree is an unweighted, directed, acyclic graph (DAG).
  - Each node's in-degree is at most 1, and out-degree is at most 2.
  - There is exactly one path from the root to every node.
- A *linked list* is also a graph:
  - Unweighted DAG.
  - In/out degree of at most 1 for all nodes.


$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \arrow[r] & D \end{tikzcd}$$


$$\begin{tikzcd} & & F \arrow[dl] \arrow[drr] & & \\ & B \arrow[dl] \arrow[d] & & & K \arrow[dl] \\ A & E & & H \arrow[dl] \arrow[d] & \\ & & G & J & \end{tikzcd}$$

Searching for paths

- Searching for a path from one vertex to another:
  - Sometimes, we just want *any* path (or want to know there *is* a path).
  - Sometimes, we want to minimize path *length* (# of edges).
  - Sometimes, we want to minimize path *cost* (sum of edge weights).
- What is the shortest path from MIA to SFO? Which path has the minimum cost?



GRAPHS

Definition. A ***directed graph* (*digraph*)** $G = (V, E)$ is an ordered pair consisting of

- a set V of ***vertices*** (singular: ***vertex***),
- a set $E \subseteq V \times V$ of ***edges***.

In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices.

In either case, we have $|E| = O(V^2)$. Moreover, if G is connected, then $|E| \geq |V| - 1$, which implies that $\lg|E| = \Theta(\lg V)$.

(Review CLRS, Appendix B.)

Adjacency-matrix representation

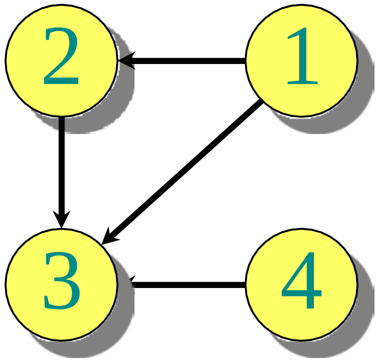
The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,.\,.\, n, 1 \,.\,.\, n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$

Adjacency-matrix representation

The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,.\,.\, n, 1 \,.\,.\, n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$

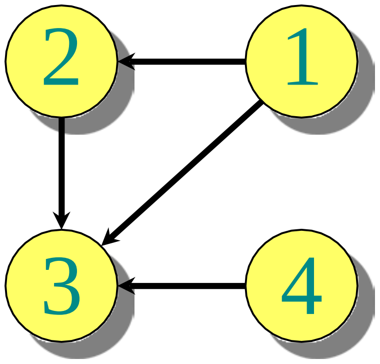


$$\begin{array}{c|cccc} A & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

$\Theta(V^2)$ storage $\Rightarrow$ ***dense*** representation.

ADJACENCY-LIST REPRESENTATION

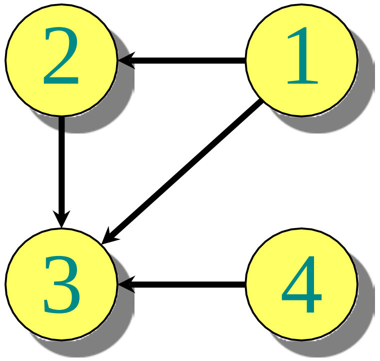
An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

ADJACENCY-LIST REPRESENTATION

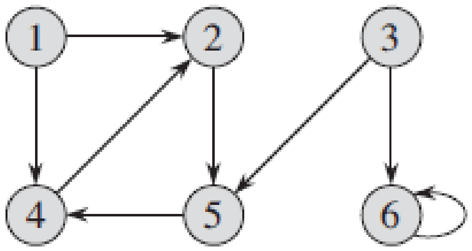
An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.

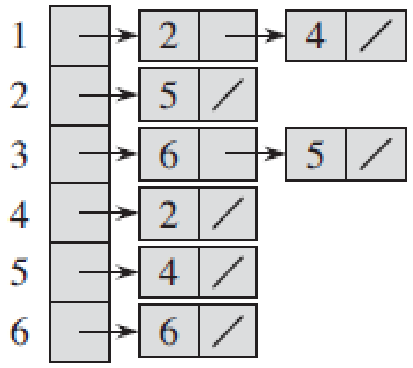


$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

Adjacency-list-Matrix -Example

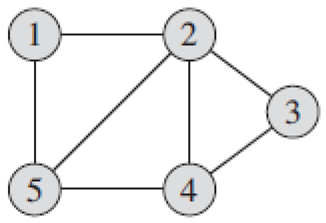

(a)


(b)

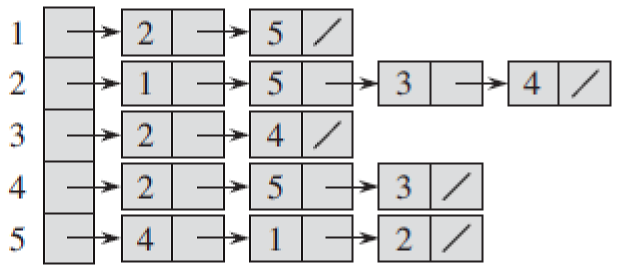
$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}$$

(c)

Adjacency-list-Matrix -Example



(a)



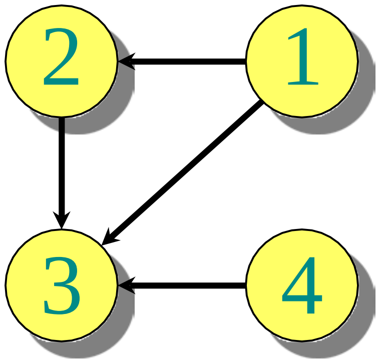
(b)

$$\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}$$

(c)

ADJACENCY-LIST REPRESENTATION

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

Handshaking Lemma: $\sum_{v \in V} degree(v) = 2|E|$ for undirected graphs $\Rightarrow$ adjacency lists use $\Theta(V + E)$ storage — a ***sparse*** representation.

Application of Graphs
Using a model to solve a complicated traffic light problem

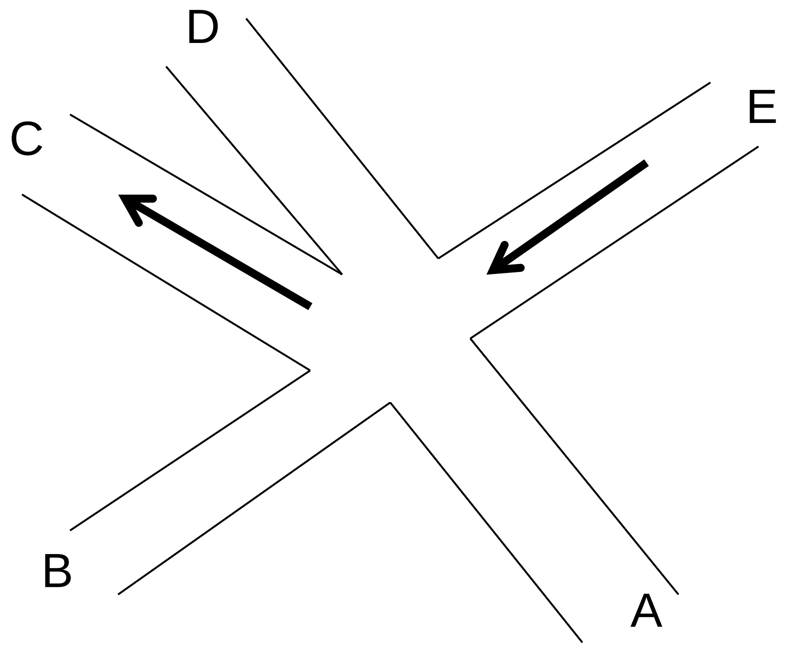
GIVEN: A complex intersection.

OBJECTIVE: Traffic light with minimum phases.

SOLUTION:

- Identify permitted turns, going straight is a “turn”.
- Make group of permitted turns.
- Make the smallest possible number of groups.
- Assign each phase of the traffic light to a group.

- Using a model to solve a complicated traffic light problem



An intersection

- Using a model to solve a complicated traffic light problem

Roads C and E are one way, others are two way.

There are 13 permitted turns.

Some turns such as AB (from A to B) and EC can be carried out simultaneously.

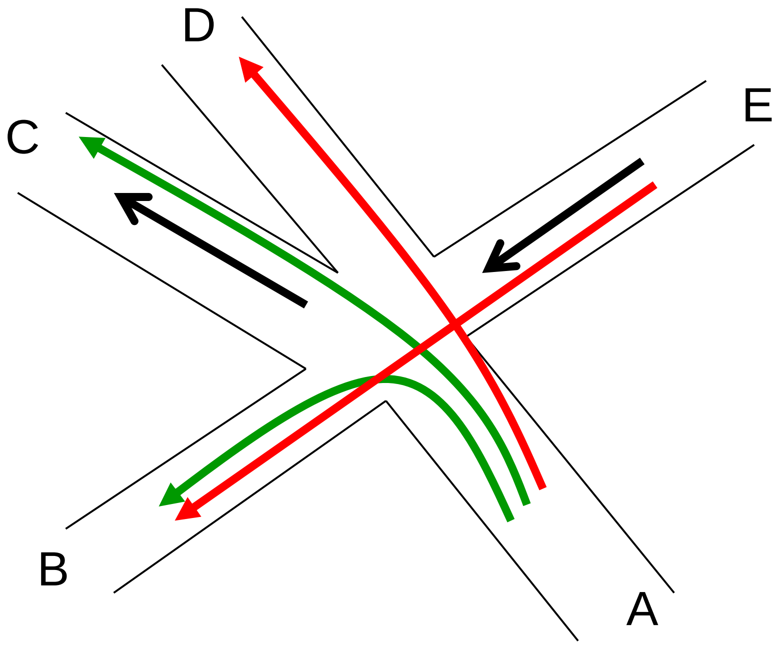
Other like AD and EB cross each other and can not be carried out simultaneously.

The traffic light should permit AB and EC simultaneously, but should not allow AD and EB.

- Using a model to solve a complicated traffic light problem

AB & AC

~~AD & EB~~



An intersection

Using a model to solve a complicated traffic light problem

SOLUTION:

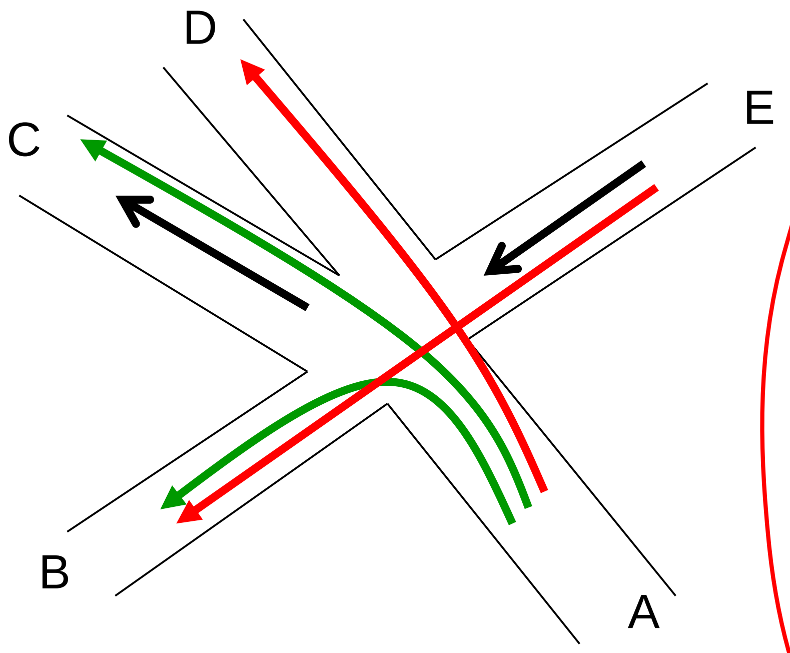
• We model the problem using a structure called graph *G(V,E)*.

• A graph consists of a set of points called *vertices*, and lines connecting the points, called *edges*.

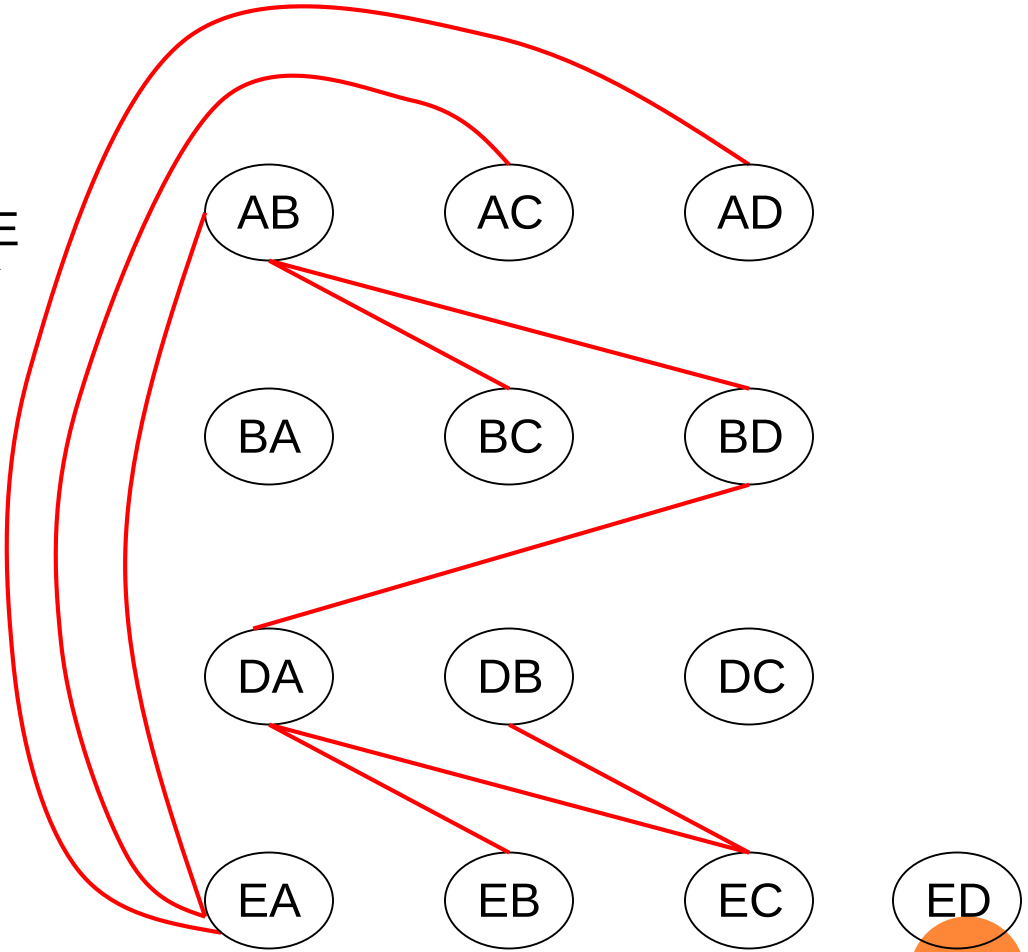
•Drawing a graph such that the vertices represent turns.

• Edges between those turns that can NOT be performed simultaneously.

- Using a model to solve a complicated traffic light problem



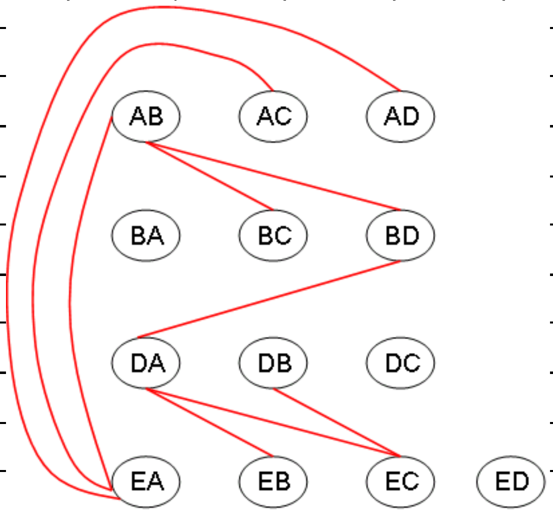
An intersection



Partial graph of incompatible turns

Using a model to solve a complicated traffic light problem

	AB	AC	AD	BA	BC	BD	DA	DB	DC	EA	EB	EC	ED
AB					1	1	1			1			
AC						1	1			1	1		
AD										1	1	1	
BA													
BC													
BD													
DA													
DB													
DC													
EA													
EB													
EC													
ED													



Partial table of incompatible turns

- Using a model to solve a complicated traffic light problem

SOLUTION:

The graph can aid in solving our problem.

A *coloring* of a graph is an assignment of a color to each vertex of the graph, so that no two vertices connected by an edge have the same color.

Our problem is of coloring the graph of incompatible turns using as few colors as possible.

- Using a model to solve a complicated traffic light problem

More on SOLUTION (Graph coloring):

The problem has been studied for decades.

The theory of algorithms tells us a lot about it.

Unfortunately this belongs to a class of problems called as **NP-Complete problems**.

- Using a model to solve a complicated traffic light problem

An algorithm that quickly produces good but not necessarily optimal solutions is called a *heuristic*.

A reasonable heuristic for graph coloring is the *greedy* algorithm.

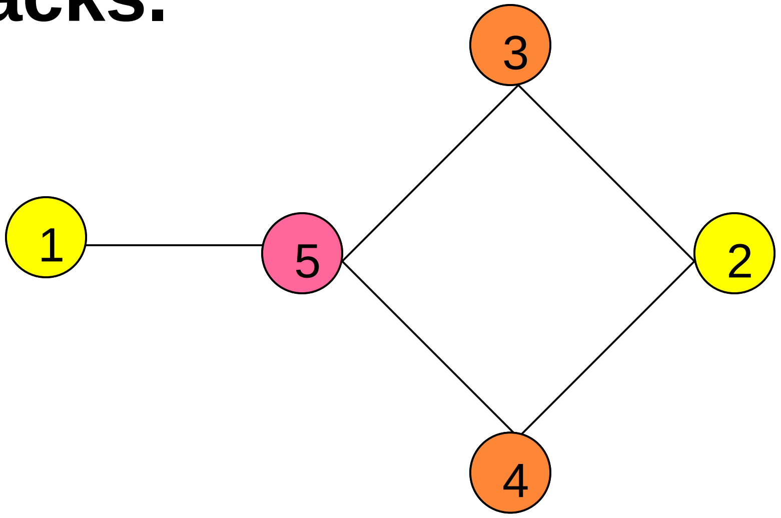
Try to color as many vertices as possible with the first color, and then as many uncolored vertices with the second color, and so on.

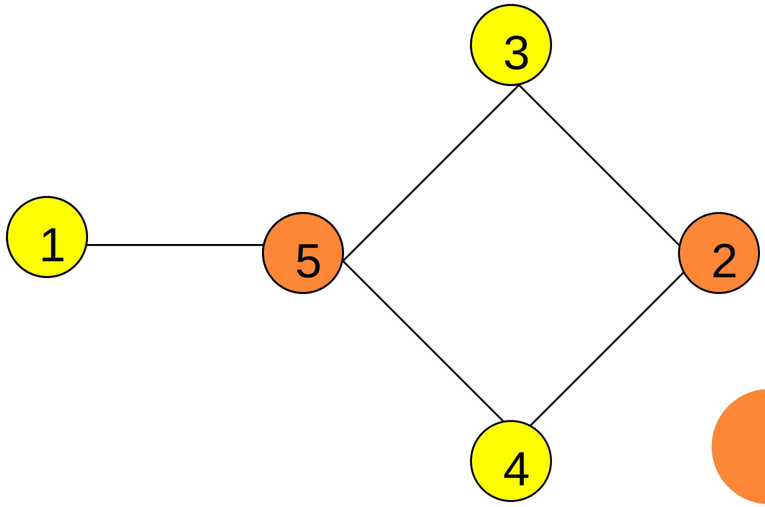
The approach would be:

1. Select some uncolored vertex, and color with new color.

2. Scan the list of uncolored vertices. For each uncolored vertex, determine whether it has an edge to any vertex already colored with the new color. If there is no such edge, color the present vertices with the new color.

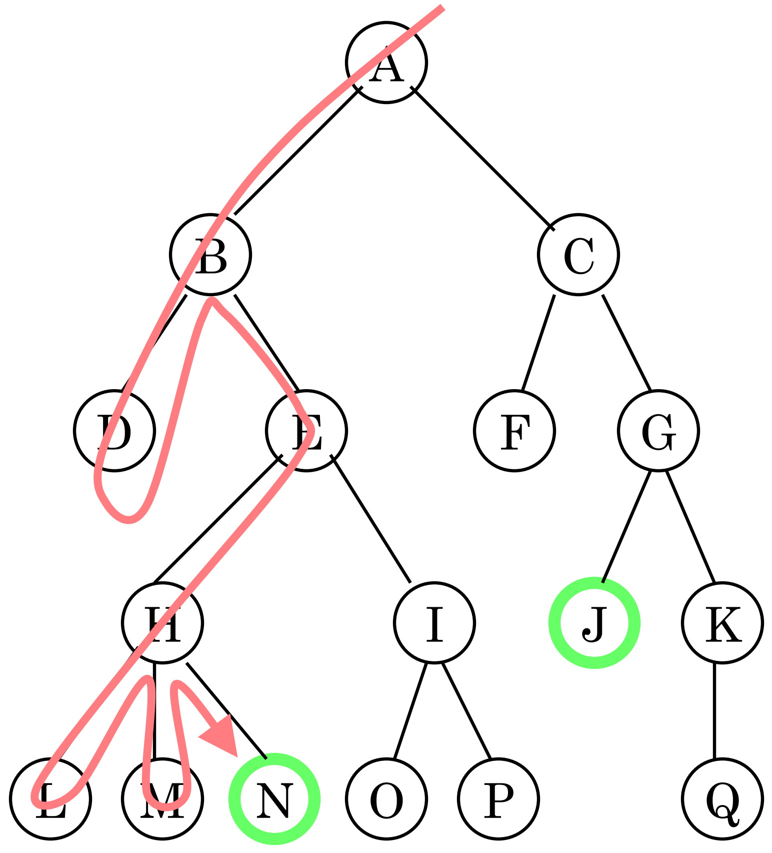
This is called “greedy”, because it colors a vertex, whenever it can, without considering potential drawbacks.





Graph Traversals

Depth-first searching

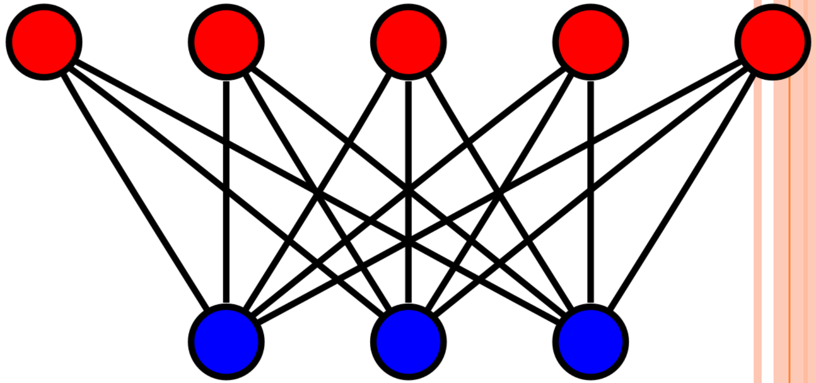


- A **depth-first** search **(DFS)** explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q
- N will be found before J

Depth First Search

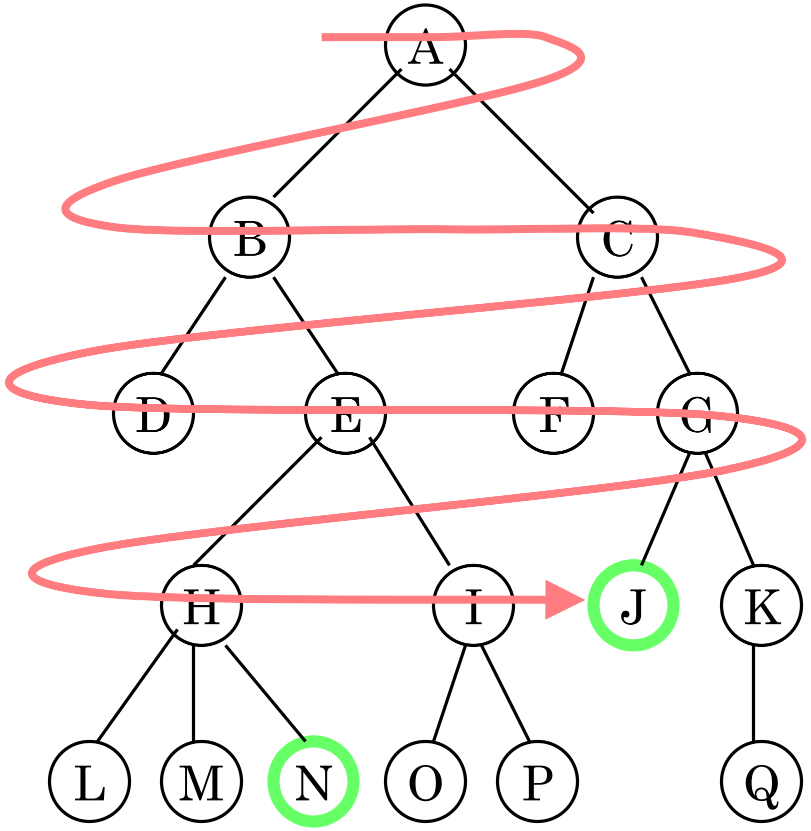
Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

- **Detecting cycle in a graph**
- **Path Finding**
- **Topological Sorting**
- **To test if a graph is bipartite**
- **Finding Strongly Connected Components of a graph**
- **Solving puzzles with only one solution**



Bipartite Graph

Breadth-first searching



- A **breadth-first** search **(BFS)** explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N

BREADTH-FIRST SEARCH

- ***Breadth-first search*** is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms.
- Given a graph G = (V, E) and a distinguished ***source*** **vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s.**
- It computes the distance (smallest number of edges) from s to each reachable vertex.
- It also produces a "breadth-first tree" with root s that contains all reachable vertices.
- For any vertex reachable from s, the simple path in the breadth-first tree from s to corresponds to a "shortest path" from s to in G, that is, a path containing the smallest number of edges.
- The algorithm works on both directed and undirected graphs.

Breadth-first searching

- **Shortest Path and Minimum Spanning Tree for unweighted graph**
- **Peer to Peer Networks**
- **Crawlers in Search Engines**
- **Social Networking Websites**
- **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
- **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
- **In Garbage Collection**
- **Cycle detection in undirected graph:**

Breadth-first searching

- To test if a graph is Bipartite
- Path Finding
- Finding all nodes within one connected component.