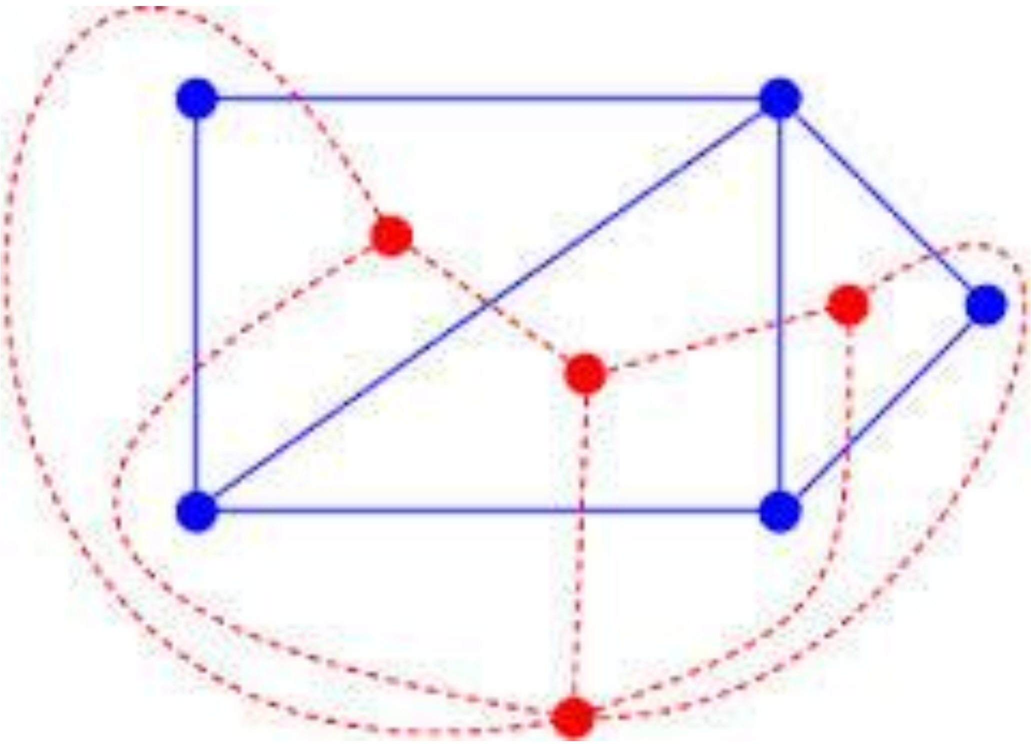
Graphs

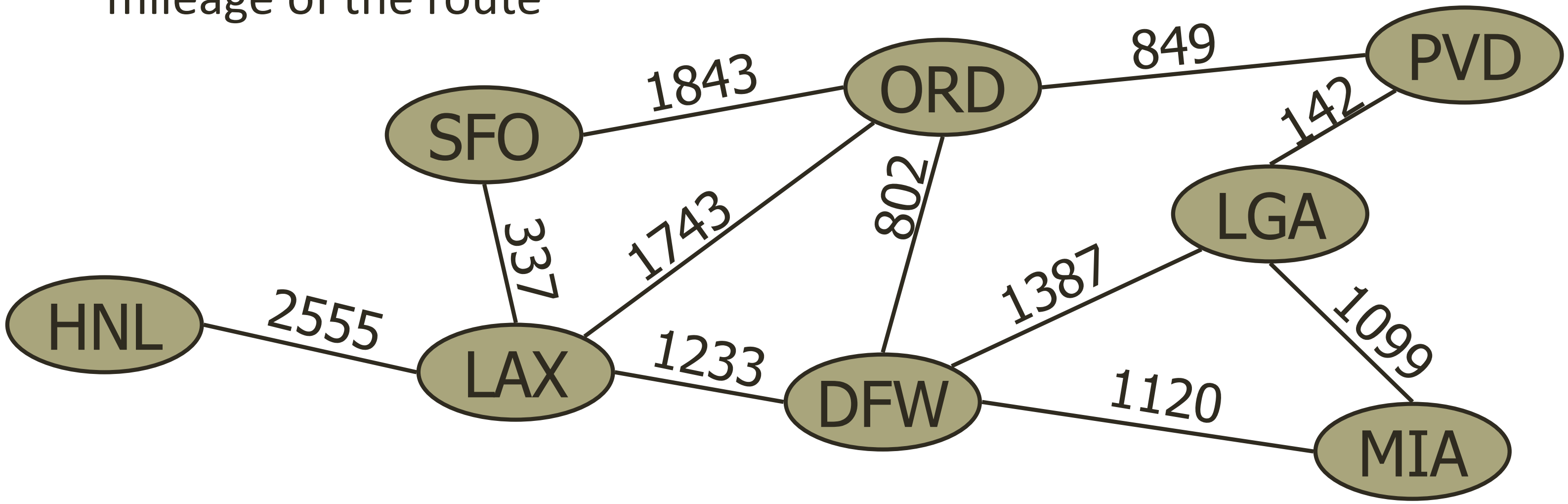


Overview

- Graph terminology & representation data structures
- Traversals:
  - Depth First Search
  - Breadth First Search
- Topological Sorting of DAGs
- Transitive Closure: Floyd-Warshall
- Shortest Paths in Weighted graphs
  - Dijkstra, Bellman-Ford, DAGs
- Minimum Spanning Trees
  - Prim-Jarnik, Kruskal
- Union-Find Structures

Graphs

- A graph is a pair (***V, E***), where
  - ***V*** is a set of nodes, called vertices
  - ***E*** is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- **Example:**
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



Graph applications

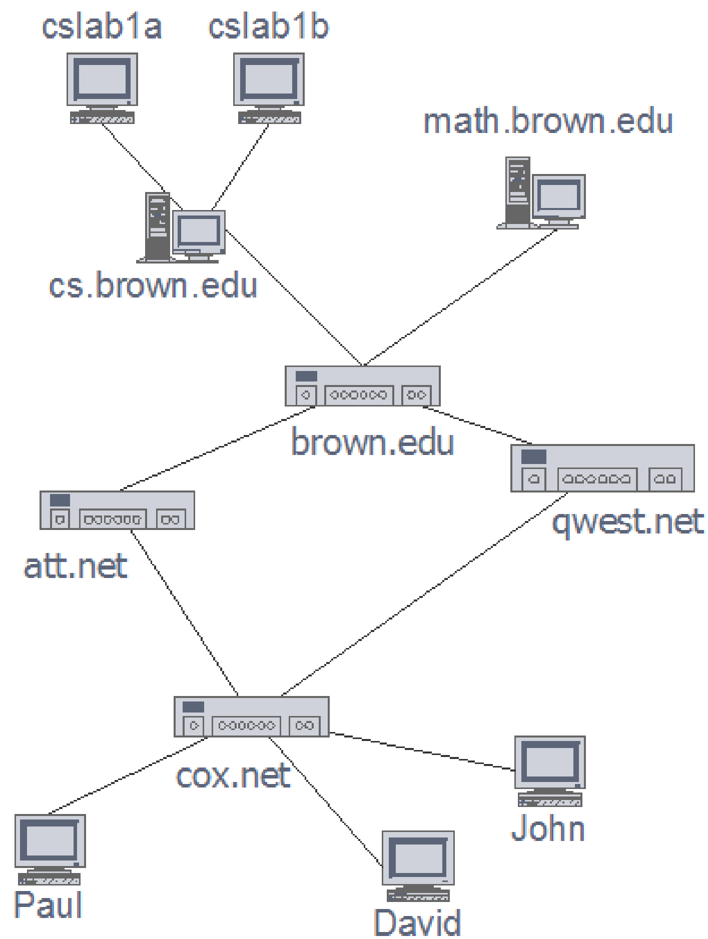
graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	intersection	street
internet	class C network	connection
game	board position	legal move
social relationship	person	friendship
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

Edge Types

- **Directed edge**
  - ordered pair of vertices (***u***,***v***)
  - first vertex ***u*** is the origin
  - second vertex ***v*** is the destination
  - e.g., a flight
- **Undirected edge**
  - unordered pair of vertices (***u***,***v***)
  - e.g., a flight route
- **Directed graph**
  - all the edges are directed
  - e.g., route network
- **Undirected graph**
  - all the edges are undirected
  - e.g., flight network

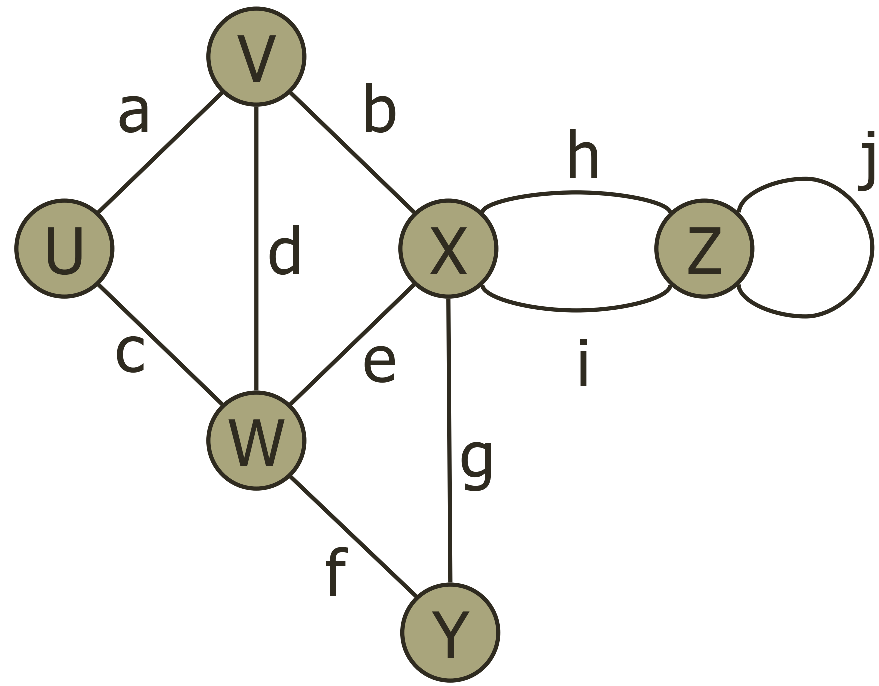
Applications

- **Electronic circuits**
  - Printed circuit board
  - Integrated circuit
- **Transportation networks**
  - Highway network
  - Flight network
- **Computer networks**
  - Local area network
  - Internet
  - Web
- **Databases**
  - Entity-relationship diagram



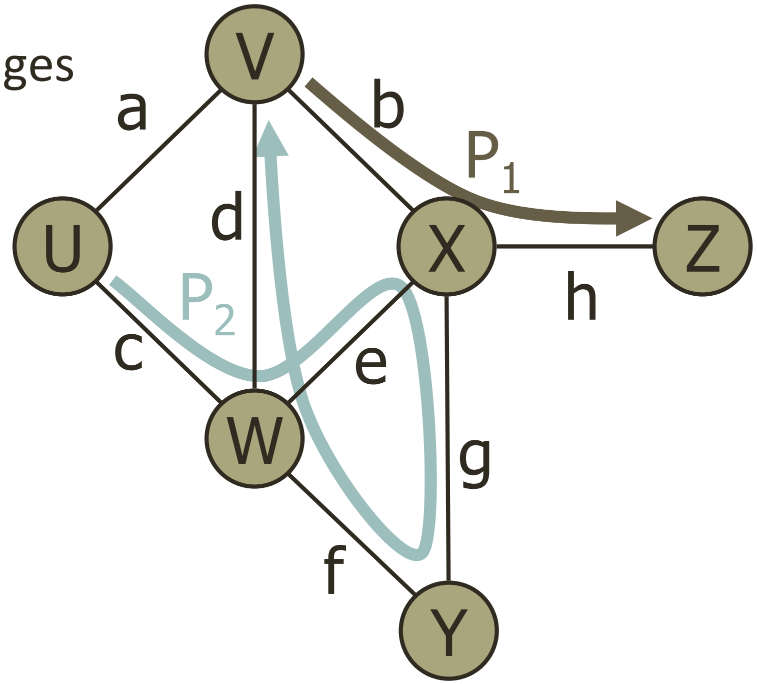
Terminology

- **End vertices (or endpoints) of an edge**
  - U and V are the endpoints of a
- **Edges incident on a vertex**
  - a, d, and b are incident on V
- **Adjacent vertices**
  - U and V are adjacent
- **Degree of a vertex**
  - X has degree 5
- **Parallel edges**
  - h and i are parallel edges
- **Self-loop**
  - j is a self-loop



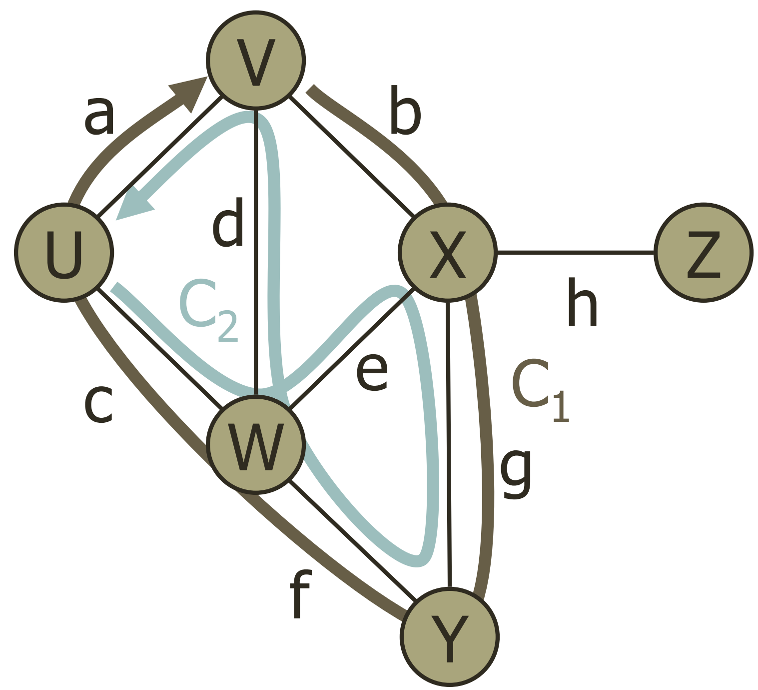
Terminology (cont.)

- **Path**
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- **Simple path**
  - path such that all its vertices and edges are distinct
- **Examples:**
  - $P_1 = (V,b,X,h,Z)$ is a simple path
  - $P_2 = (U,c,W,e,X,g,Y,f,W,d,V)$ is a path that is not simple



Terminology (cont.)

- **Cycle**
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- **Simple cycle**
  - cycle such that all its vertices and edges are distinct
- **Examples:**
  - C_1 = (V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2 = (U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple



Properties

Notation

n	number of vertices
m	number of edges
deg(v)	degree of vertex v

Property 1

$\sum_v \deg(v) = 2m$

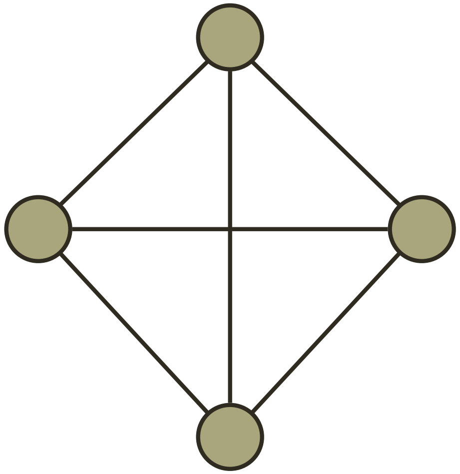
Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

$m \leq n\,(n - 1)/2$

Proof: each vertex has degree at most (n - 1)



Vertices and Edges

- A **graph** is a collection of **vertices** and **edges**.
- We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.
- A **Vertex** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code)
  - We assume it supports a method, element(), to retrieve the stored element.
- An **Edge** stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element( ) method.

Graph ADT: part 1

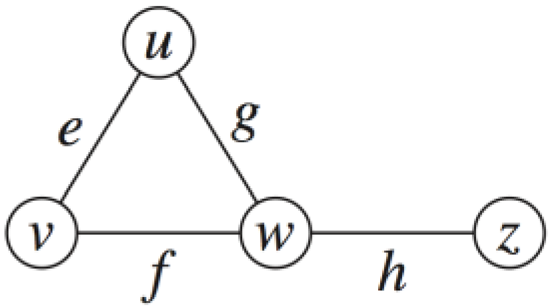
numVertices( ):	Returns the number of vertices of the graph.
vertices( ):	Returns an iteration of all the vertices of the graph.
numEdges( ):	Returns the number of edges of the graph.
edges( ):	Returns an iteration of all the edges of the graph.
getEdge(u, v):	Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
endVertices(e):	Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
opposite(v, e):	For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.

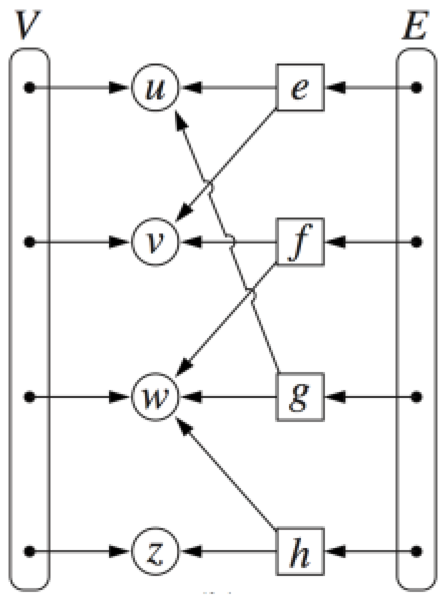
Graph ADT: part 2

outDegree(v):	Returns the number of outgoing edges from vertex v.
inDegree(v):	Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).
outgoingEdges(v):	Returns an iteration of all outgoing edges from vertex v.
incomingEdges(v):	Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).
insertVertex(x):	Creates and returns a new Vertex storing element x.
insertEdge(u, v, x):	Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
removeVertex(v):	Removes vertex v and all its incident edges from the graph.
removeEdge(e):	Removes edge e from the graph.

Edge List Structure

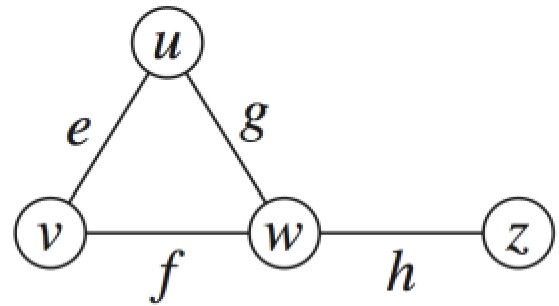
- **Vertex object**
  - element
  - reference to position in vertex sequence
- **Edge object**
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- **Vertex sequence**
  - sequence of vertex objects
- **Edge sequence**
  - sequence of edge objects

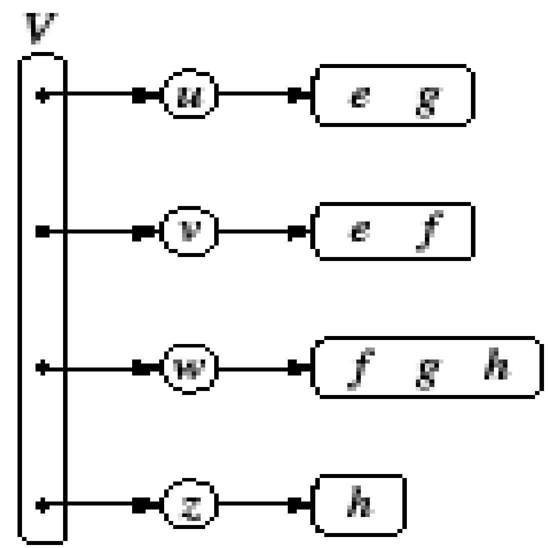




Adjacency List Structure

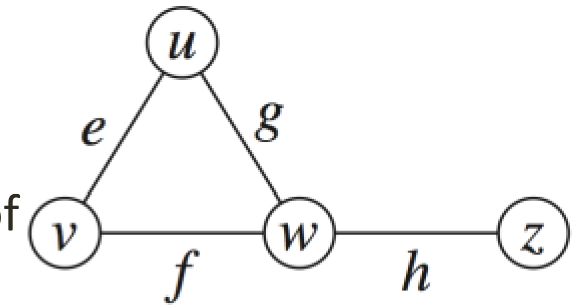
- **Incidence sequence for each vertex**
  - sequence of references to edge objects of incident edges
- **Augmented edge objects**
  - references to associated positions in incidence sequences of end vertices



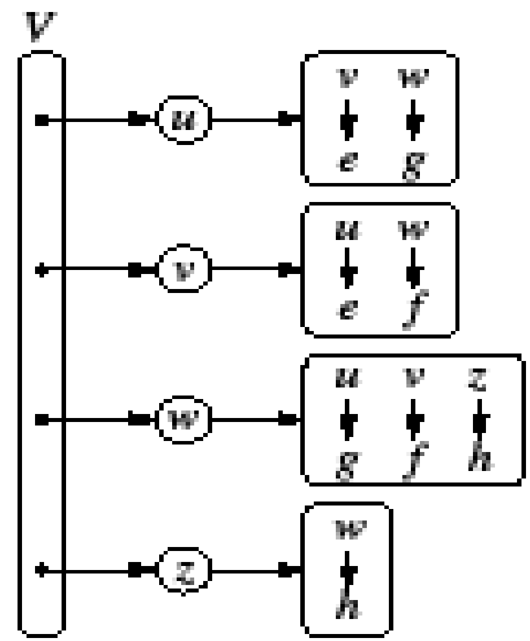


Adjacency Map Structure

- **Incidence sequence for each vertex**
  - sequence of references to adjacent vertices, each mapped to edge object of the incident edge

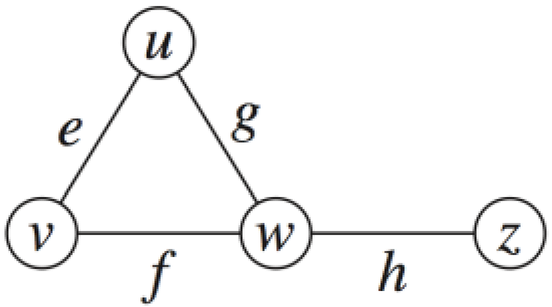


- **Augmented edge objects**
  - references to associated positions in incidence sequences of end vertices

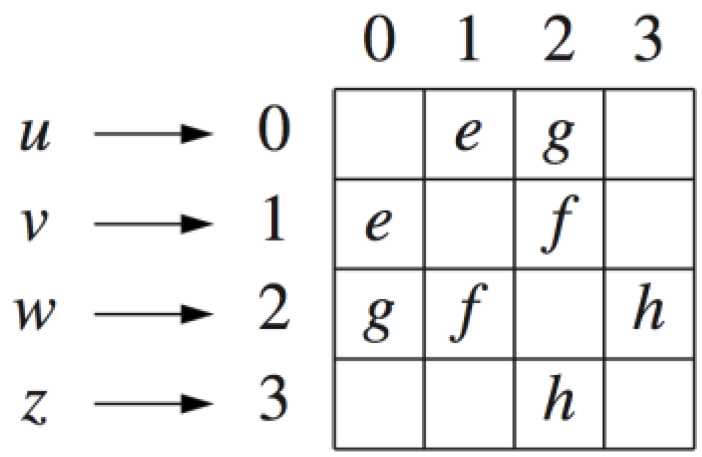


Adjacency Matrix Structure

- **Edge list structure**
- **Augmented vertex objects**
  - Integer key (index) associated with vertex
- **2D-array adjacency array**
  - Reference to edge object for adjacent vertices
  - Null for non-adjacent vertices



- The "old fashioned" version just has 0 for no edge and 1 for edge

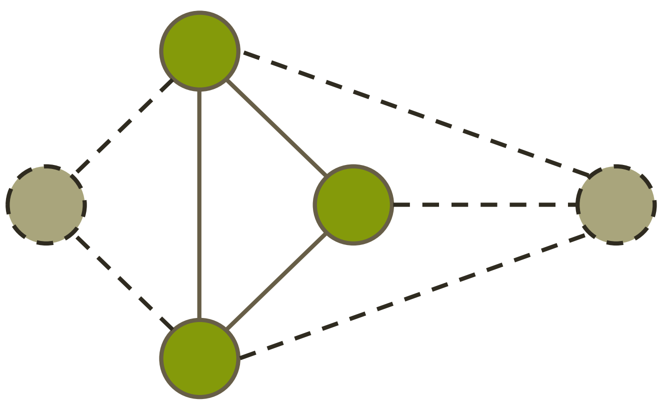


Performance

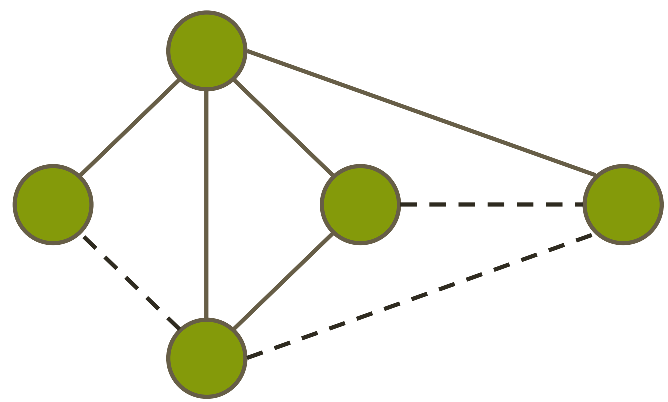
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	max(deg(v), deg(w))	1

Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A **spanning** subgraph of G is a subgraph that contains all the vertices of G



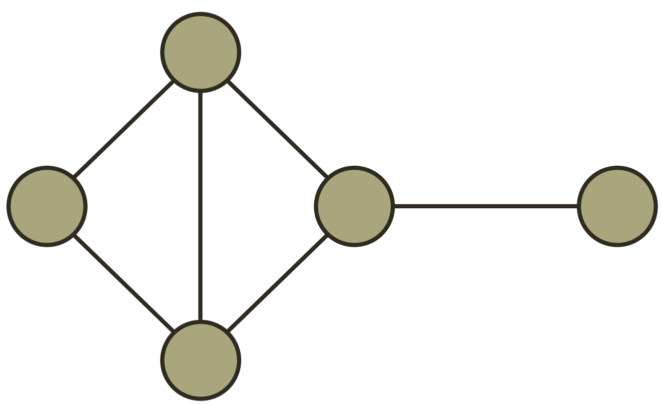
Subgraph



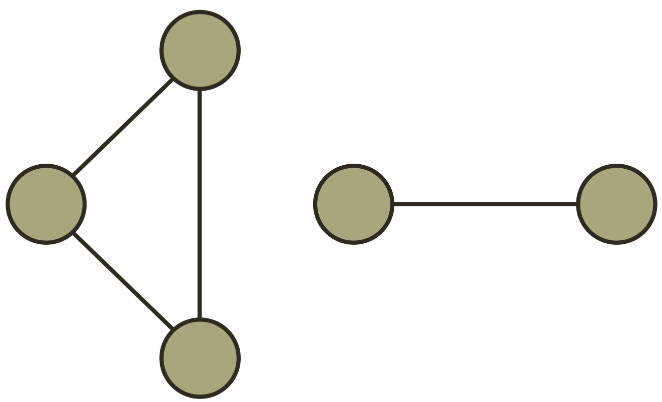
Spanning subgraph

Connectivity

- A graph is **connected** if there is a path between every pair of vertices
- A **connected component** of a graph G is a maximal connected subgraph of G



Connected graph

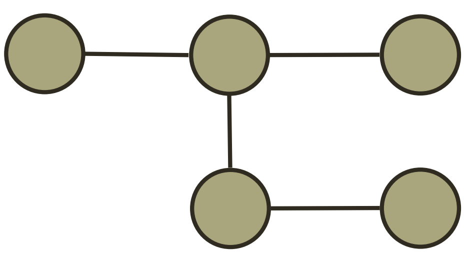


Non connected graph with two connected components

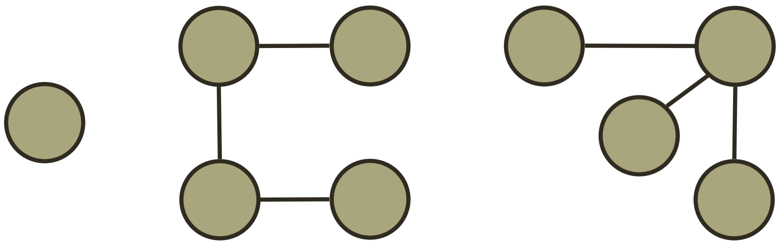
Trees and Forests

- A **(free) tree** is an undirected graph T such that
  - T is **connected**
  - T has **no cycles**

  This definition of tree is different from the one of a **rooted** tree
- A **forest** is an undirected graph without cycles
- The **connected components** of a forest are trees



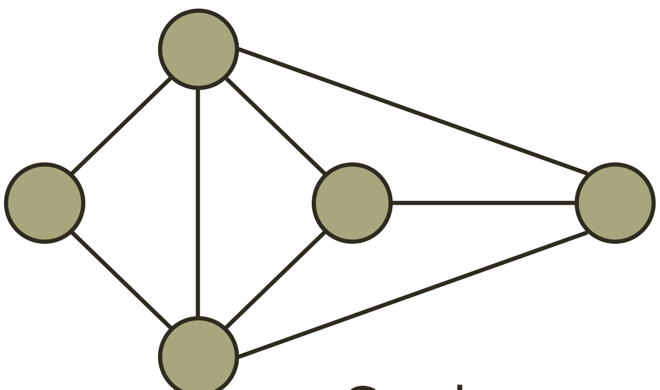
Tree



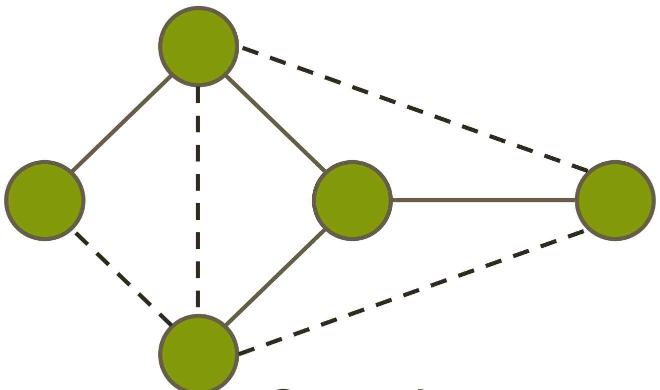
Forest

Spanning Trees and Forests

- A **spanning tree** of a connected graph is a spanning **subgraph** that is a **tree**
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

1. Depth First Search (DFS)
2. Breadth First Search (BFS)

Depth-First Search

Depth-First Search

- DFS is a general graph traversal technique
- DFS(G)
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

DFS Algorithm from a Vertex

Algorithm DFS(G, u):

Input: A graph G and a vertex u of G

Output: A collection of vertices reachable from u, with their discovery edges

Mark vertex u as visited.

for each of u's outgoing edges, $e = (u, v)$ **do**

  if vertex v has not been visited **then**

    Record edge e as the discovery edge for vertex v.

    Recursively call DFS(G, v).

- **DFS stands for Depth First Search.**
- **DFS is an algorithm for traversing or searching a tree or graph data structures.**
- **It uses a stack data structure for implementation.**
- **In DFS one starts at the root and explores as far as possible along each branch before backtracking.**

Algorithm for DFS:

[1]-- Initialize all nodes with status=1.(ready state)

[2]– Put starting node in the stack and change status to status=2(waiting state).

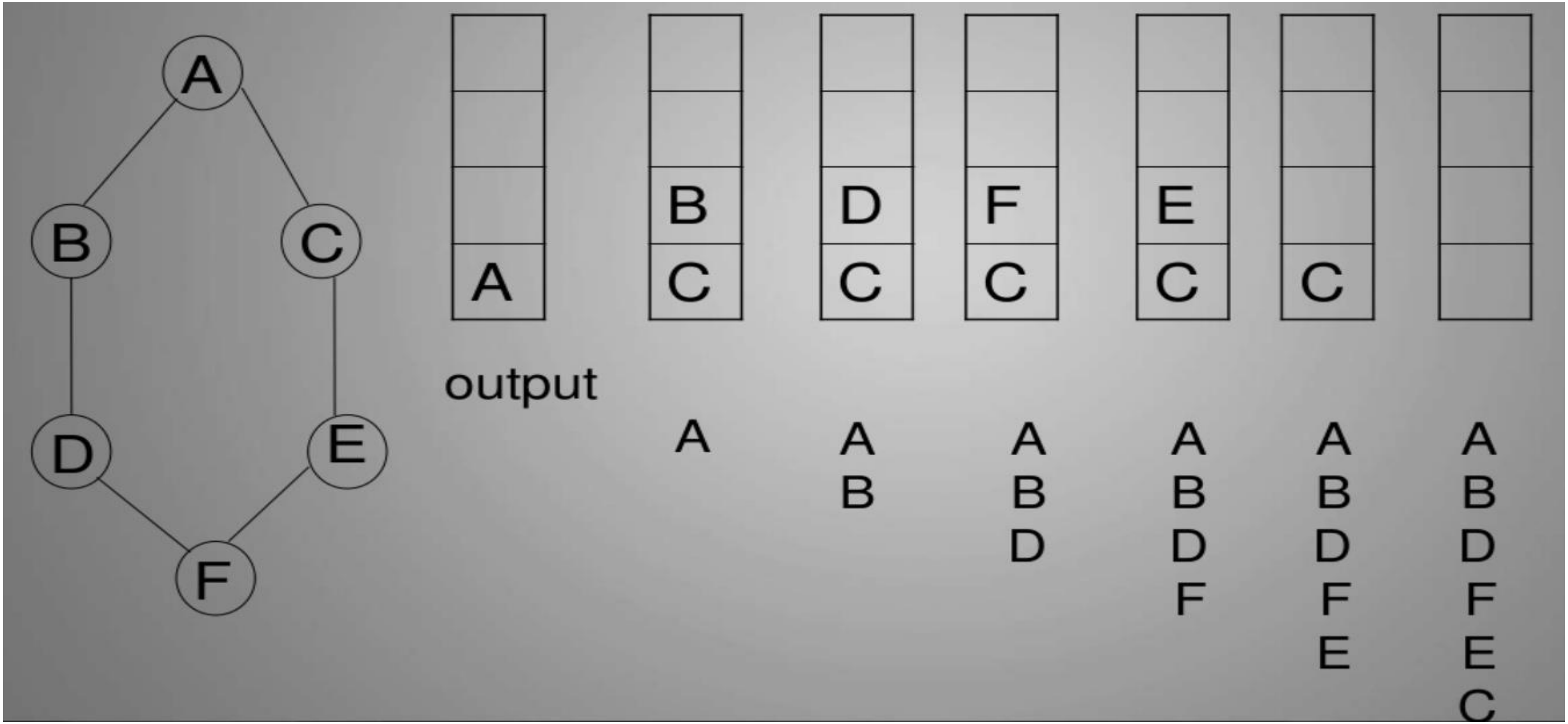
[3]– Loop:-

Repeat step- 4 and step- 5 until stack Get empty.

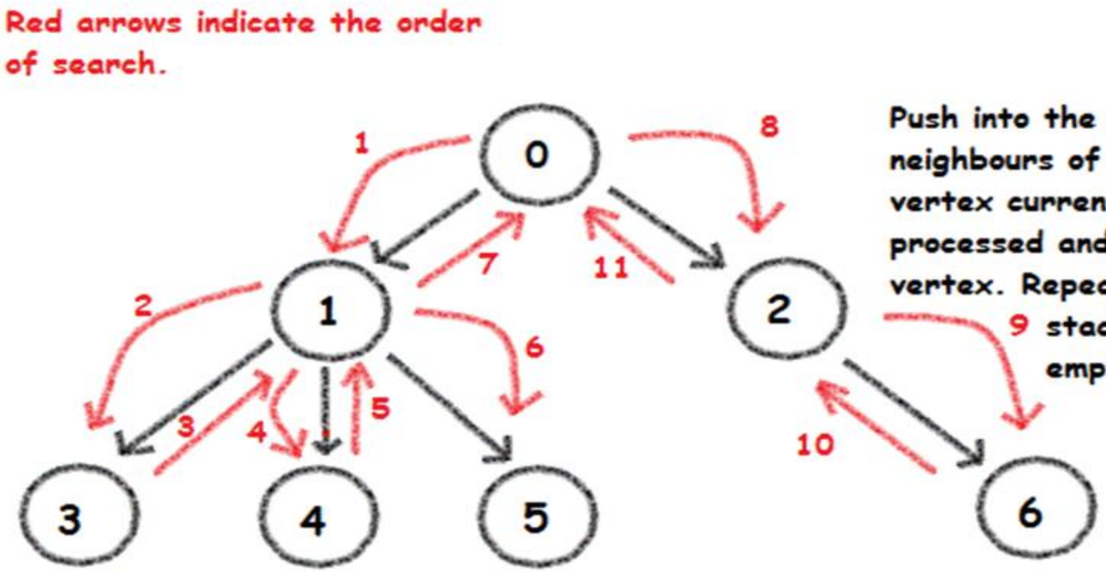
[4]– Remove top node N from stack process them and change the status of N processed state (status=3).

[5]– Add all the neighbours of N to the top of stack and change status to waiting status-2.

How DFS works:



DFS:Example



Push into the stack the neighbours of the vertex currently being processed and Pop the vertex. Repeat until stack is not empty.

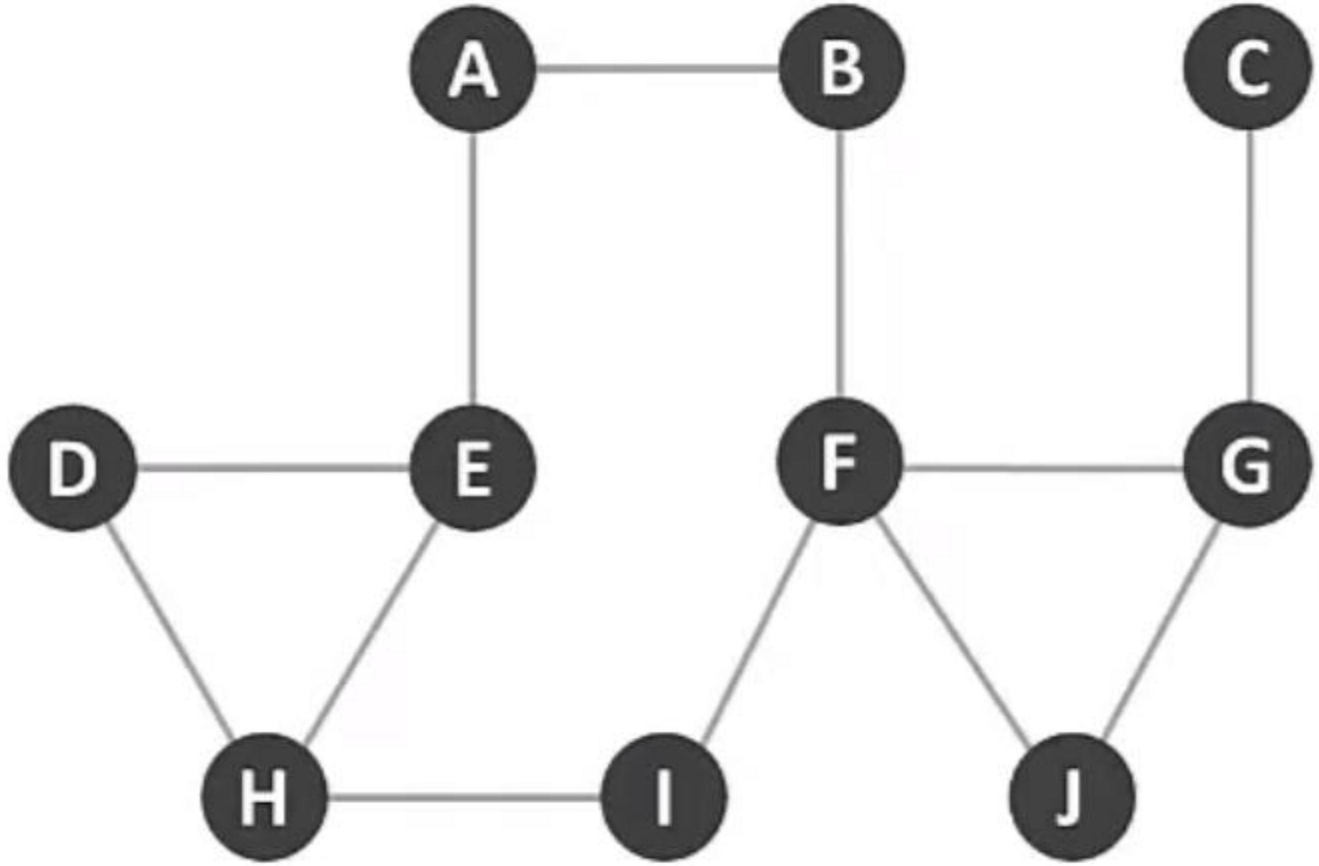
Top of stack

Vertex	Stack
	0
0	1, 2
1	3, 4, 5, 2
3	4, 5, 2
4	5, 2
5	2
2	6
6	

Depth First Search

Exercise

Apply DFS algorithm for this graph:
starting from Vertex A and going alphabetical order



Breadth-First Search

- BFS is a general graph traversal technique
- A BFS traversal of a graph G
  - Visits all vertices and edges of G
  - Determines whether G is connected
  - Computes connected components of G
  - Computes a spanning forest of G
- BFS takes $\boldsymbol{O}(\boldsymbol{n} + \boldsymbol{m})$ time
- BFS can be extended to solve other graph problems, e.g.,
  - Find a path with minimum number of edges between two given vertices
  - Minimum number of hops in a computer network (routing)
  - Find a simple cycle, if there is one

- BFS stands for <u>**B**</u>readth <u>**F**</u>irst <u>**S**</u>earch.
- BFS is an algorithm for traversing or searching a tree or graph data structures.
- It uses a queue data structure for implementation.
- In BFS traversal we visit all the nodes level by level and the traversal completed when all the nodes are visited.

Algorithm for BFS:

Step 1: Initialize all nodes with status=1.(ready state)

Step 2: Put starting node in a queue and change status to status=2.(waiting state)

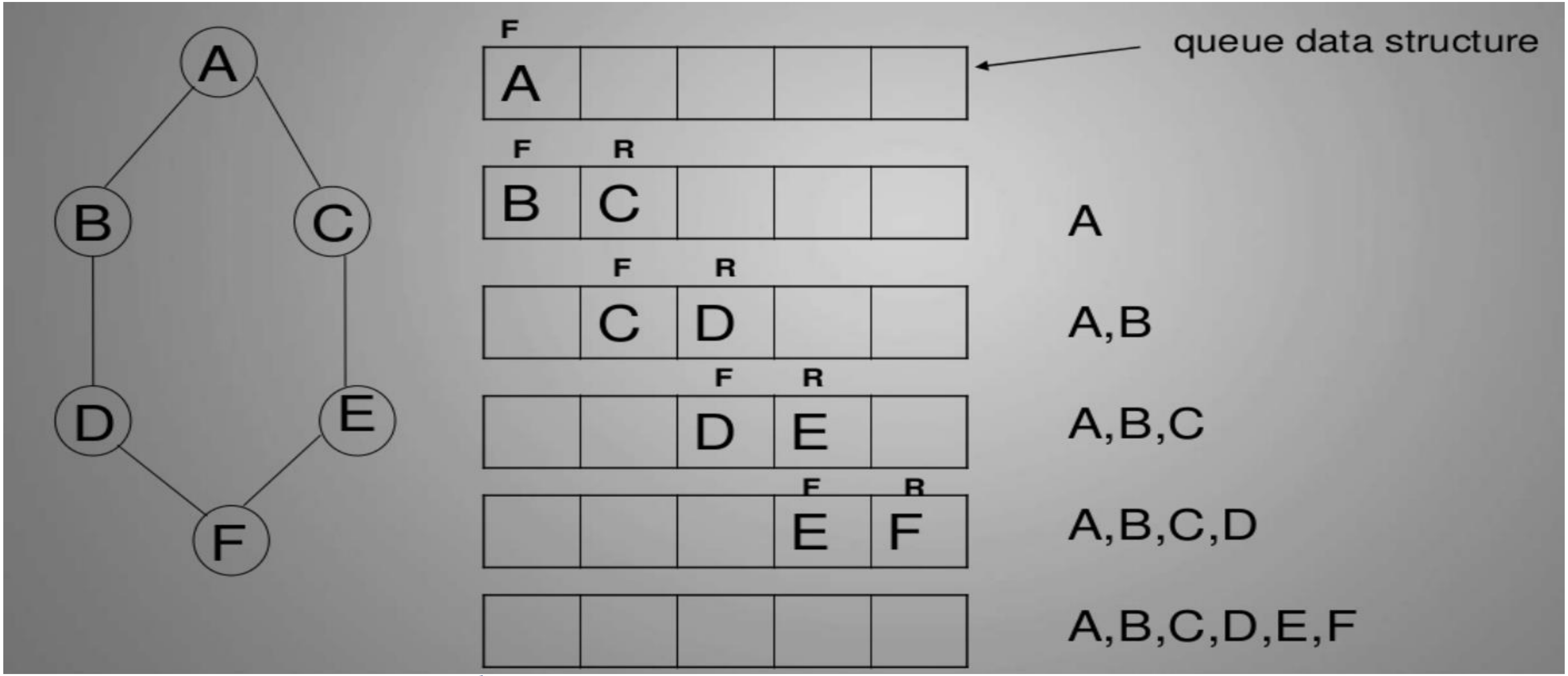
Step 3: loop:

repeat step 4 and step 5 until queue gets empty.

Step 4: Remove front node N from queue, process them and change the status of N to status=3.(processed state)

Step 5: Add all the neighbours of N to the rear of queue and change status to status=2.(waiting status)

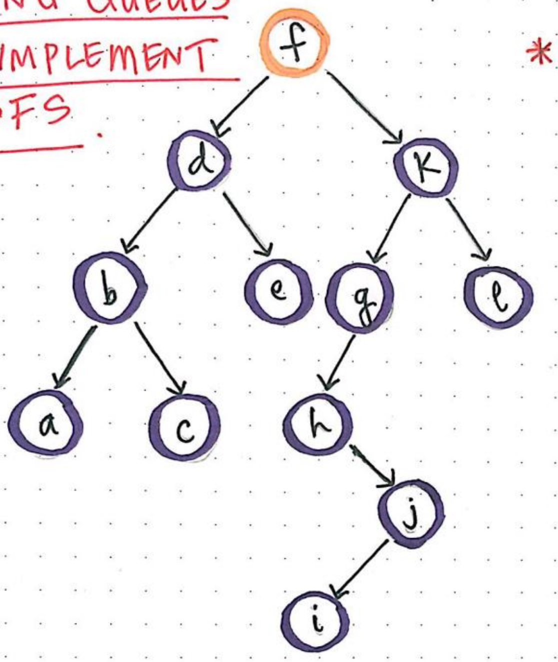
How BFS works:



Queue gets empty and the algorithm ends

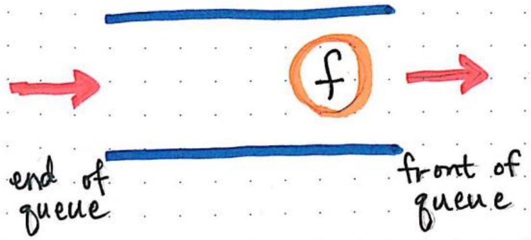
BFS:Example

USING QUEUES TO IMPLEMENT BFS.



✱ Discovered node: a node we add to our queue, whose location we know, but we haven't actually visited yet.

→ To start, the root node will be the only discovered node.

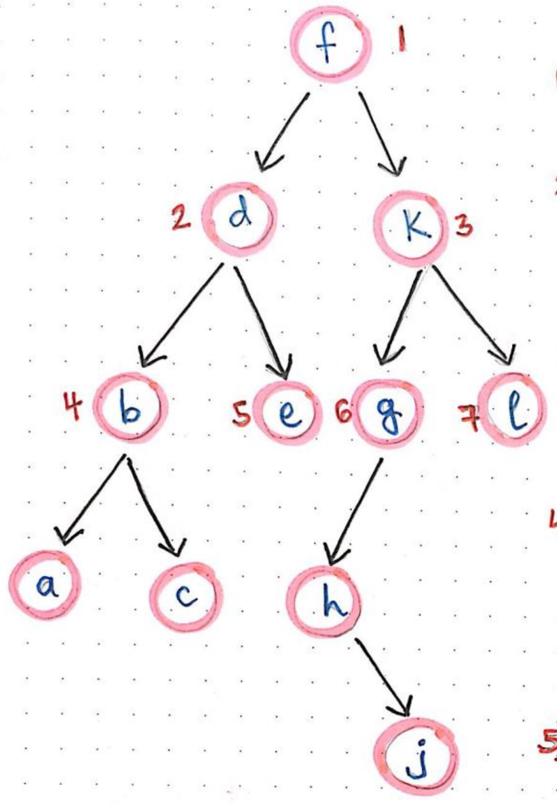


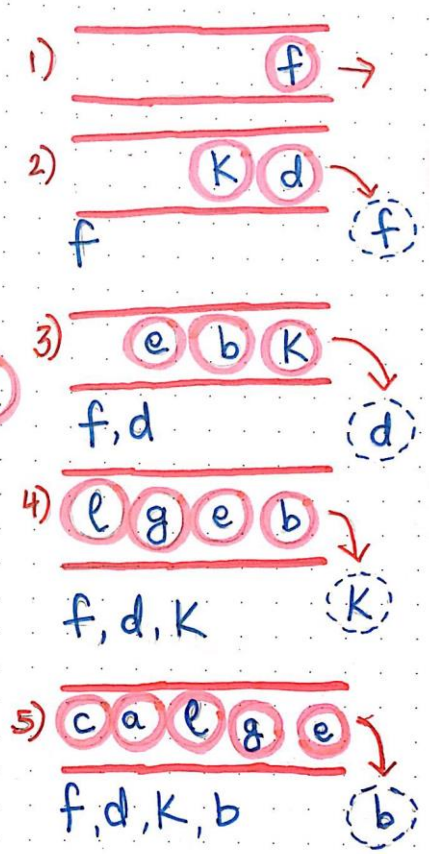
✱ Once we have at least one node enqueued (and our queue isn't empty) we can start visiting the nodes and enqueuing their children.

Steps to BFS :

1) Visit node f (print its value).
2) Enqueue left child.
3) Enqueue right child.

Continue until our queue is totally empty!





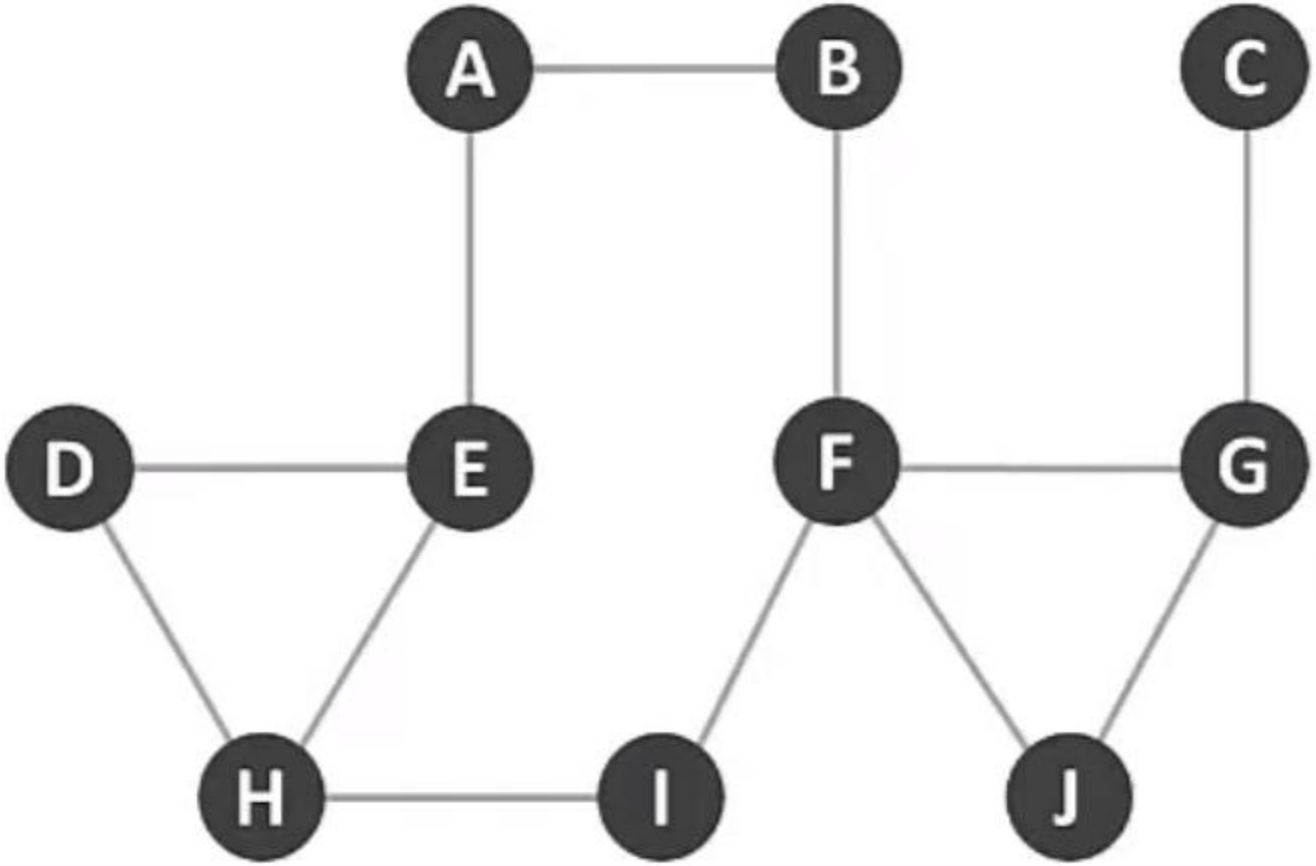
But what about space-time complexity?

→ Visiting a node (reading its data and enqueuing its children) takes constant time. Since we are only visiting each node once, the time it will take us to use a BFS is $O(n)$, where n is the number of nodes.

→ The space complexity depends on the size of the queue at its worst, which could be up to $O(n)$ also.

Exercise

Apply BFS algorithm for this graph



DFS vs. BFS

- DFS stands for Depth First Search.
- DFS can be done with the help of STACK i.e., LIOF.
- In DFS has higher time and space complexity, because at a time it needs to back tracing in graph for traversal.

- BFS stands for Breadth First Search.
- BFS can be done with the help of QUEUE i.e., FIOF.
- In BFS the space & time complexity is lesser as there is no need to do back tracing

DFS vs. BFS

- DFS is more faster then BFS.
- DFS requires less memory compare to BFS.
- DFS is not so useful in finding shortest path.
- Example :

```
      A
     / \
    B   C
   /   / \
  D   E   F
```

Ans : A,B,D,C,E,F

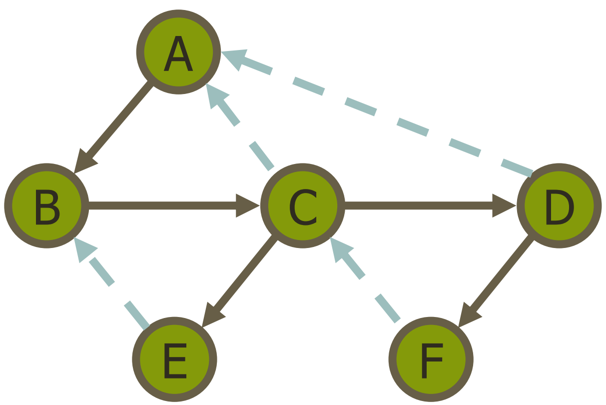
- BFS is slower than DFS.
- BFS requires more memory compare to DFS.
- BFS is useful in finding shortest path.
- Example :

```
      A
     / \
    B   C
   /   / \
  D   E   F
```

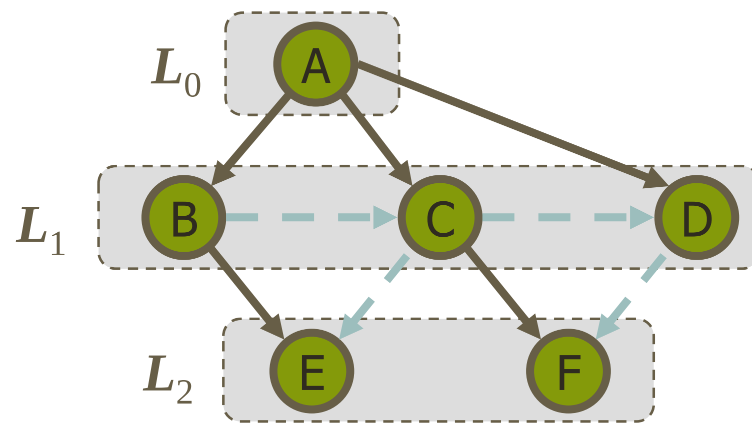
Ans : A,B,C,D,E,F

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	

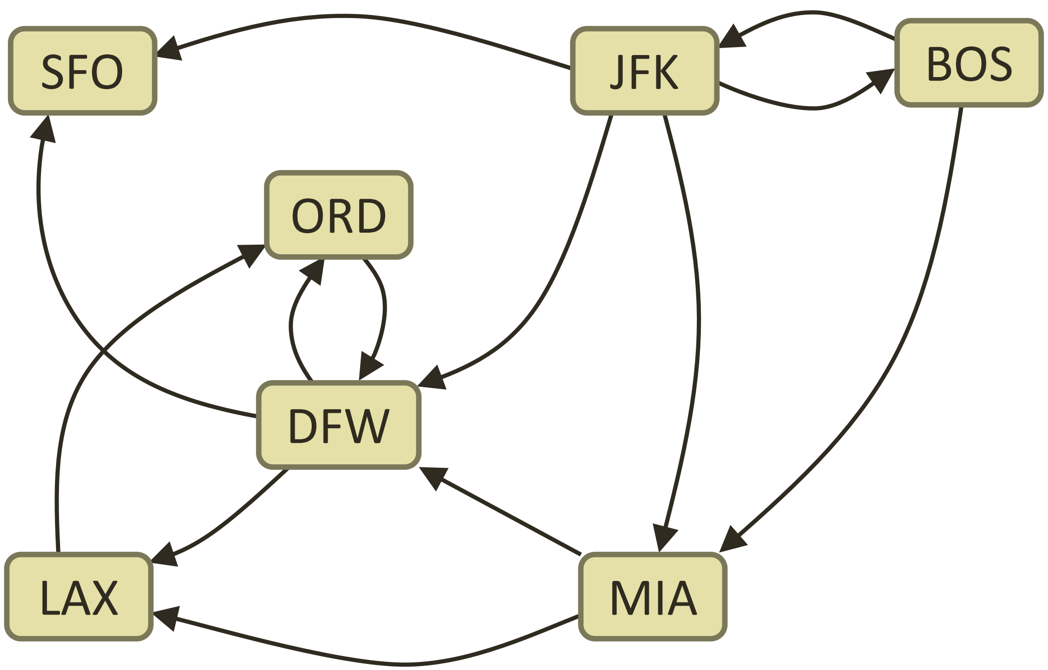


DFS



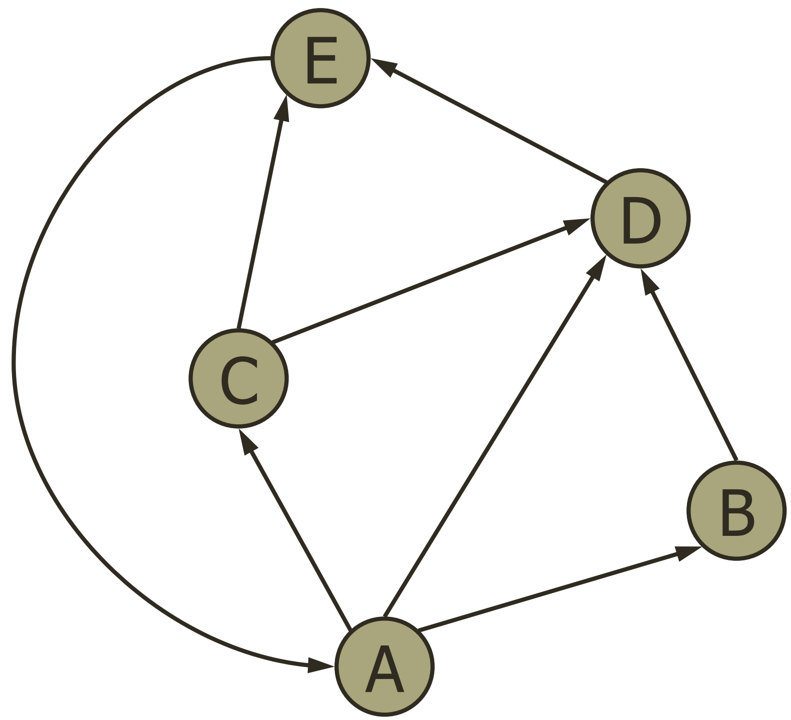
BFS

Directed Graphs

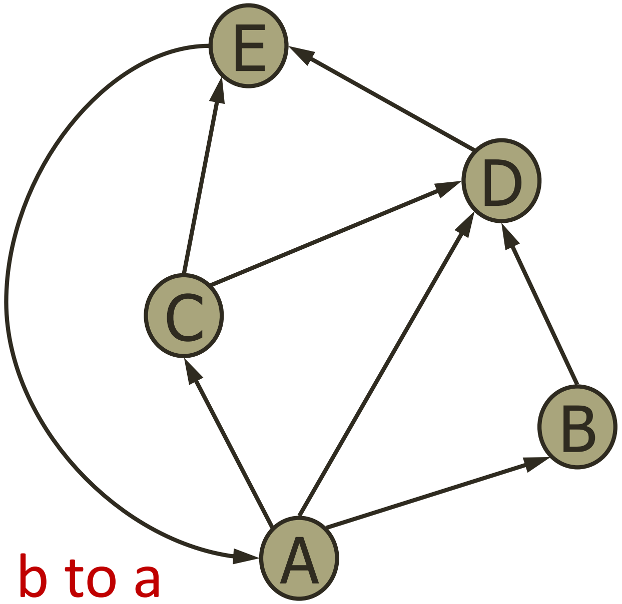


Digraphs

- A **digraph** is a graph whose edges are all directed
  - Short for "directed graph"
- **Applications**
  - one-way streets
  - flights
  - task scheduling



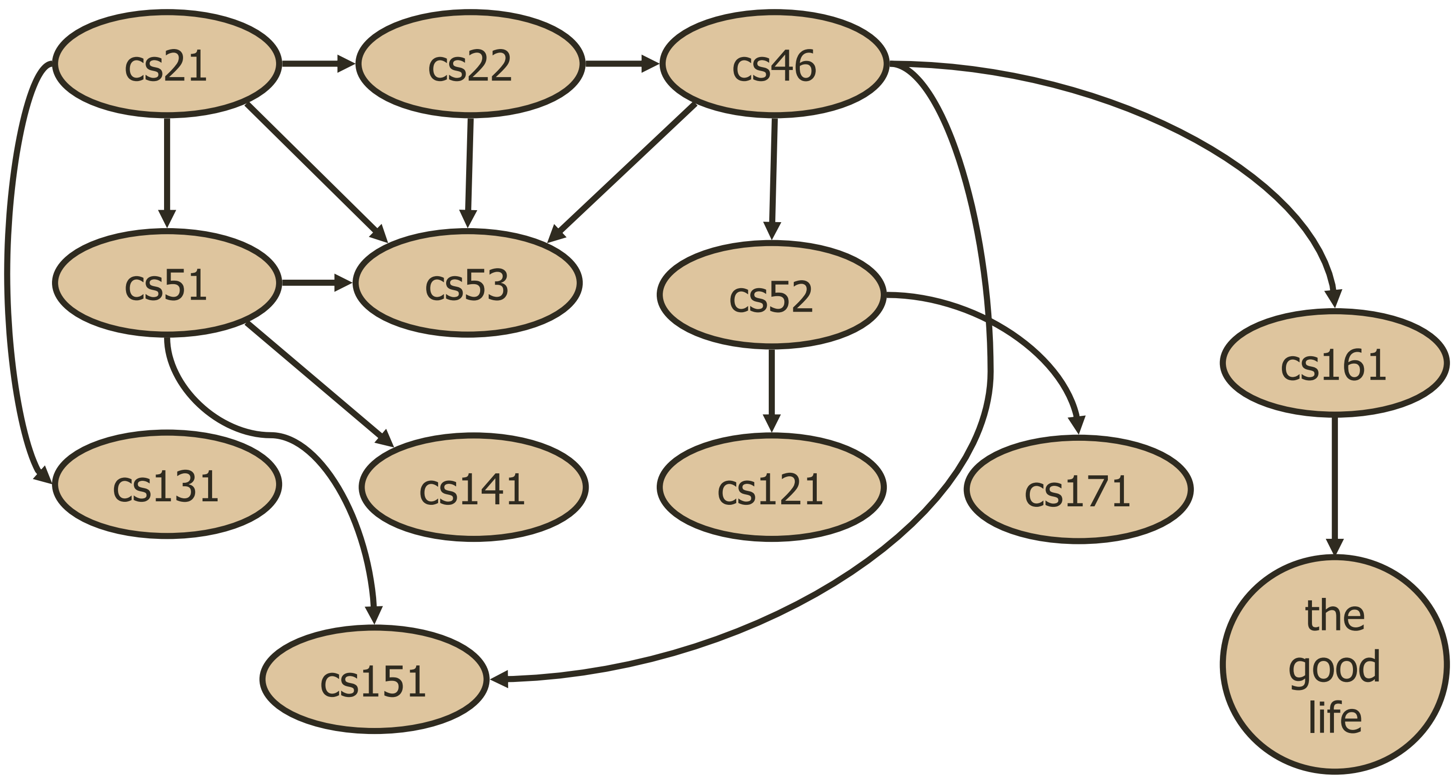
Digraph Properties



- A graph G=(V,E) such that
  - Each edge goes in one direction:
  - Edge (a,b) goes from a to b, but **not** b to a
- If G is simple, $\boldsymbol{m} \leq \boldsymbol{n} \cdot (\boldsymbol{n} - 1)$
- If we keep in-edges and out-edges in **separate** adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size

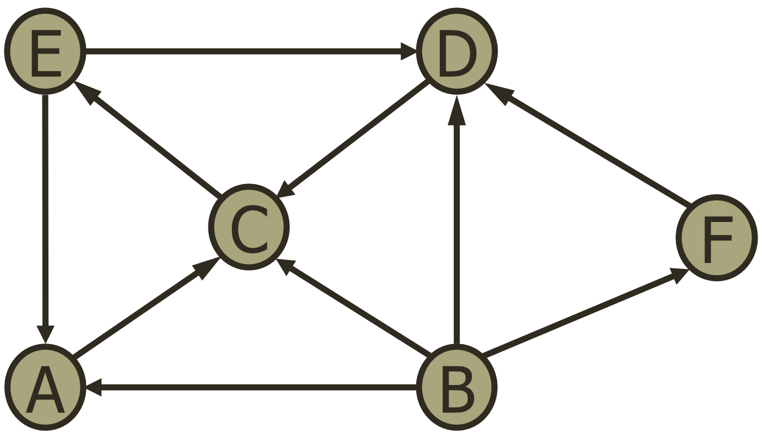
Digraph Application

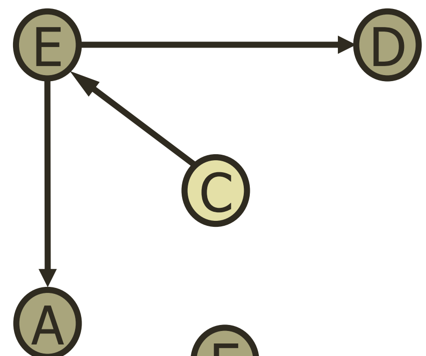
- **Scheduling:** edge (a,b) means task a must be completed before task b can be started

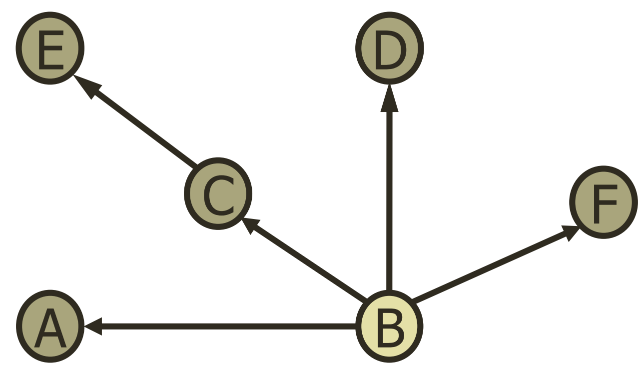


Reachability

- **DFS tree rooted at v:**
  vertices reachable from v via directed paths

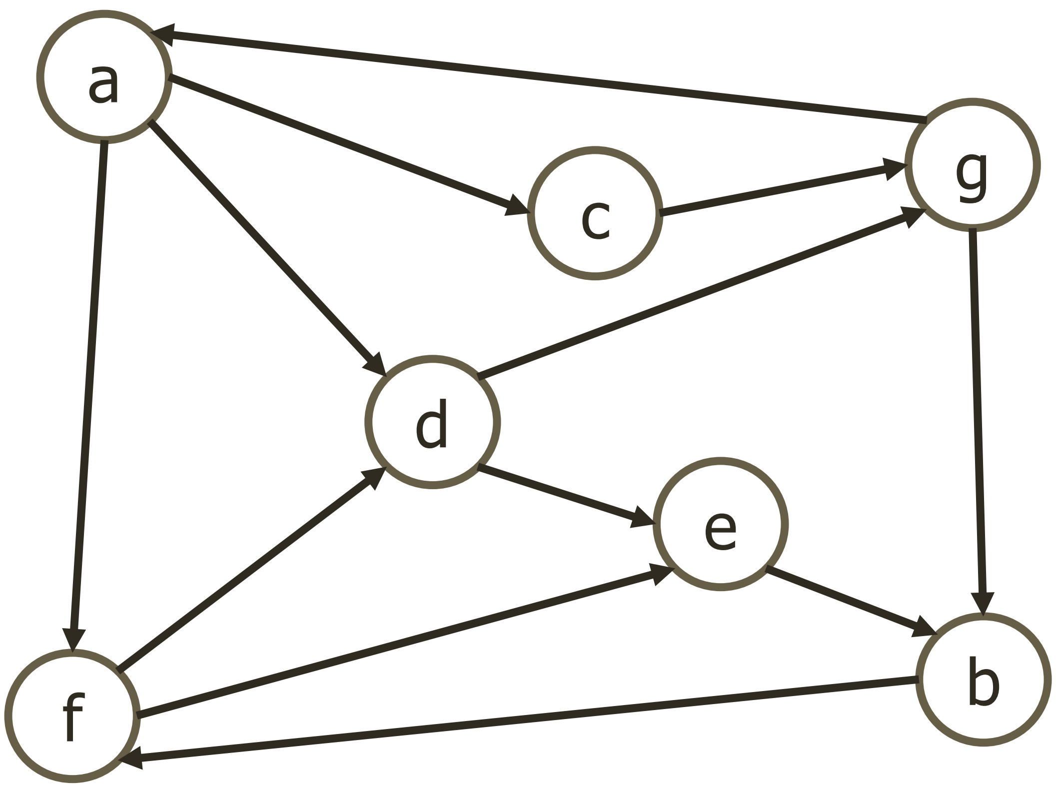




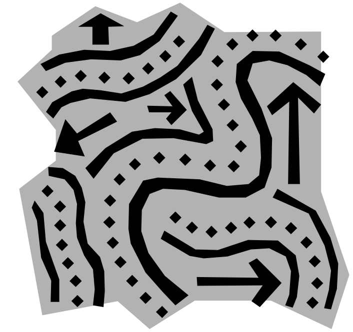


Strong Connectivity

Each vertex can reach **all** other vertices

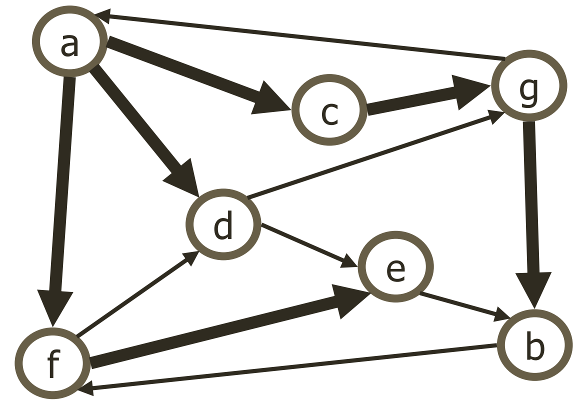


Strong Connectivity Algorithm

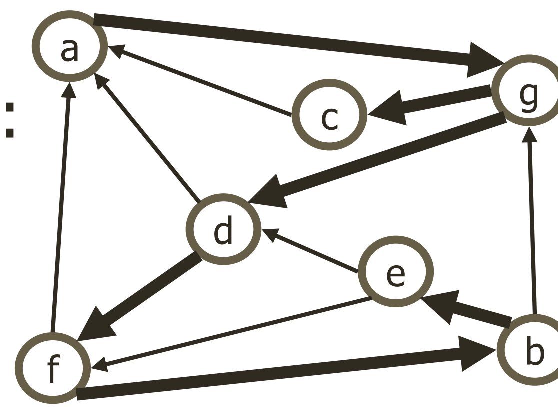


- For each vertex v in G do:
  - Perform a DFS from v in G
    - If there's a w not visited, return "no"
  - Let G' be G with edges reversed
  - Perform a DFS from v in G'
    - If there's a w not visited, return "no"
- Else, return "yes"
- Running time: O(n(n+m))

G:



G' :



3. Topological Sorting of DAGs