Graphs

Ms. Sarika Dharangaonkar,
Assistant Professor,
Information Technology Department, KJSCE

Formal definition of graphs

- A graph G is defined as follows:

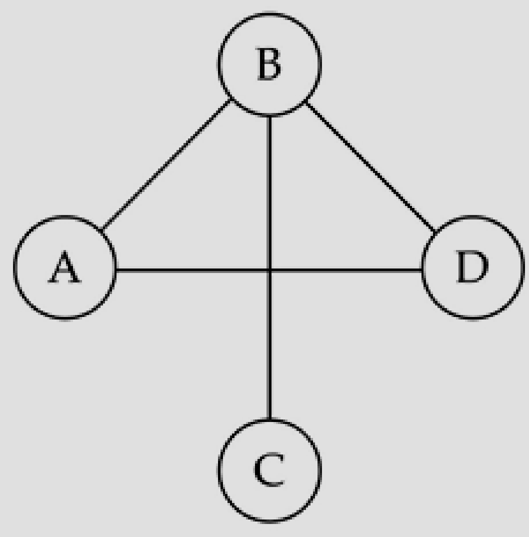
$$G=(V,E)$$

$V(G)$: a finite, nonempty set of vertices

$E(G)$: a set of edges (pairs of vertices)

Directed vs. undirected graphs

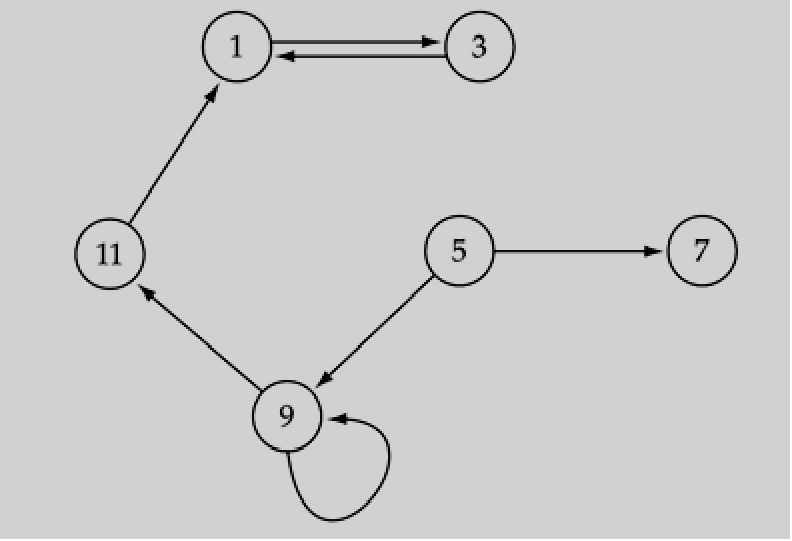
- When the edges in a graph have no direction, the graph is called *undirected*



V(Graph1) = { A, B, C, D }
E(Graph1) = { (A, B), (A, D), (B, C), (B, D) }

Directed vs. undirected graphs (cont.)

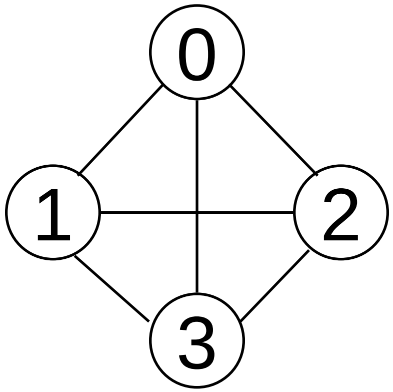
- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)



V(Graph2) = {1,3,5,7,9,11)

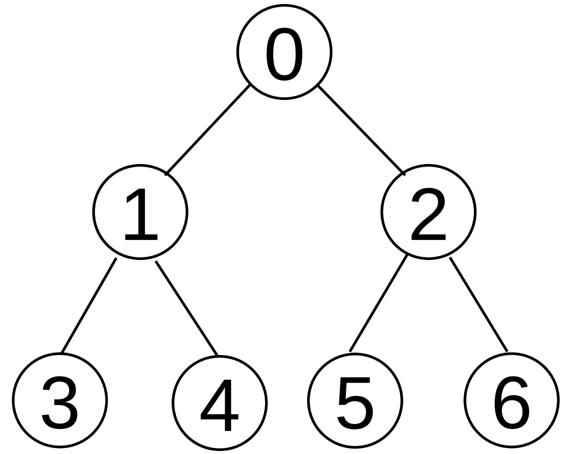
E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7) (9,9) (11,1)}

Examples for Graph



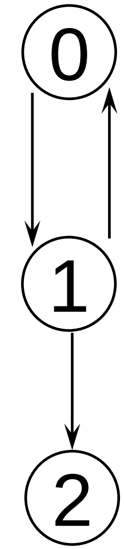
G_1

complete graph



G_2

incomplete graph



G_3

$V(G_1)=\{0,1,2,3\}$ $\quad$ $E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$

$V(G_2)=\{0,1,2,3,4,5,6\}$ $\quad$ $E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$

$V(G_3)=\{0,1,2\}$ $\quad$ $E(G_3)=\{\langle 0,1\rangle,\langle 1,0\rangle,\langle 1,2\rangle\}$

complete undirected graph: $n(n-1)/2$ edges

complete directed graph: $n(n-1)$ edges

Graph terminology

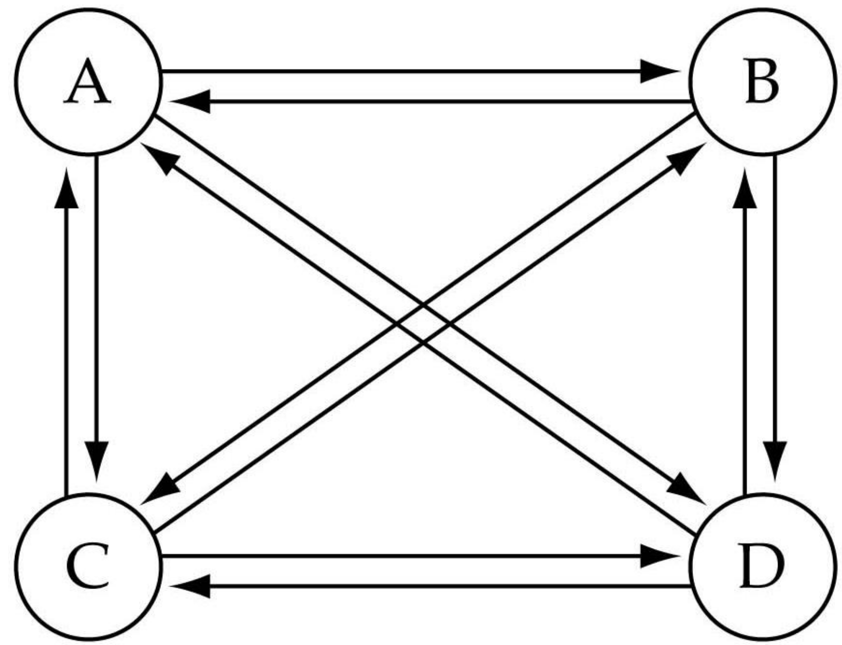
- <u>Adjacent nodes</u>: two nodes are adjacent if they are connected by an edge

- <u>Path</u>: a sequence of vertices that connect two nodes in a graph
- <u>Complete graph</u>: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

  *N * (N-1)*

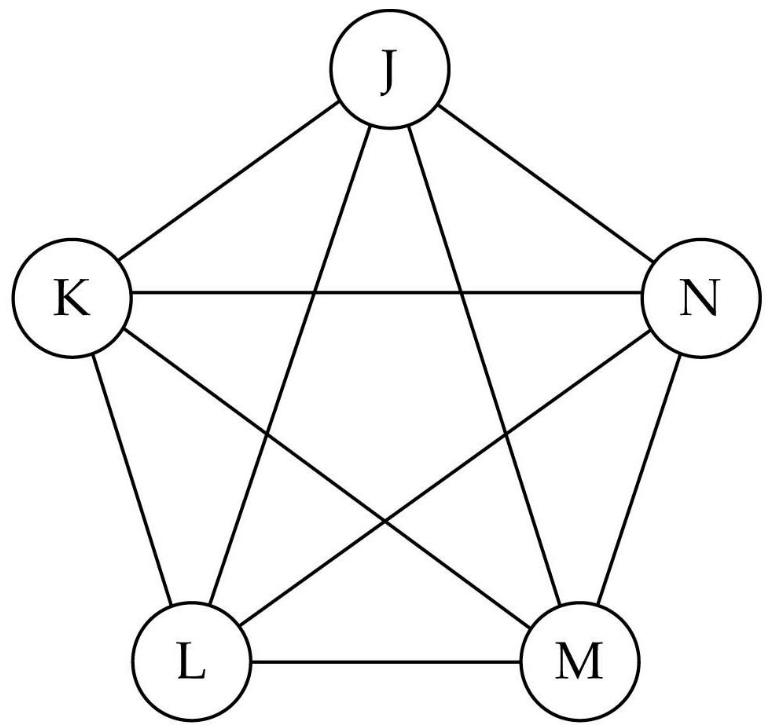


(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

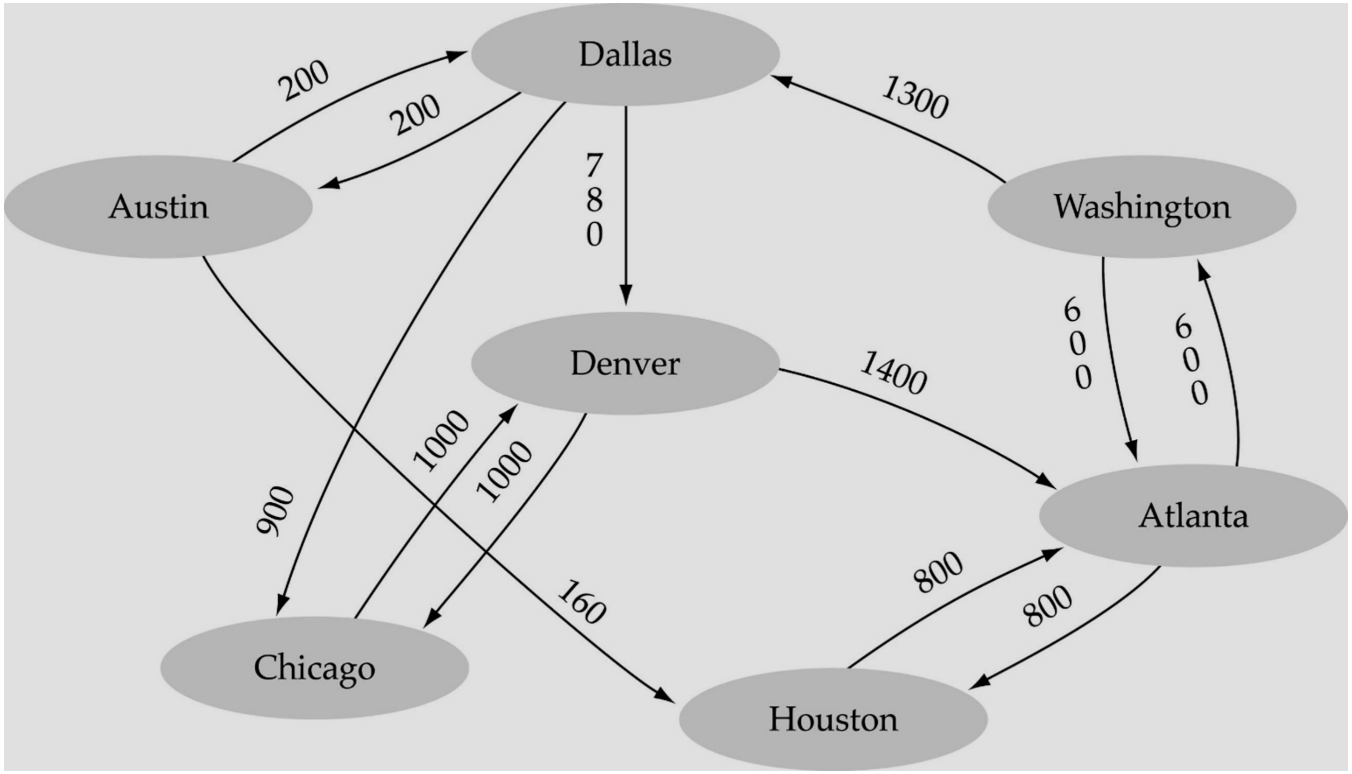
    *N * (N-1) / 2*



(b) Complete undirected graph.

Graph terminology (cont.)

- Weighted graph: a graph in which each edge carries a value

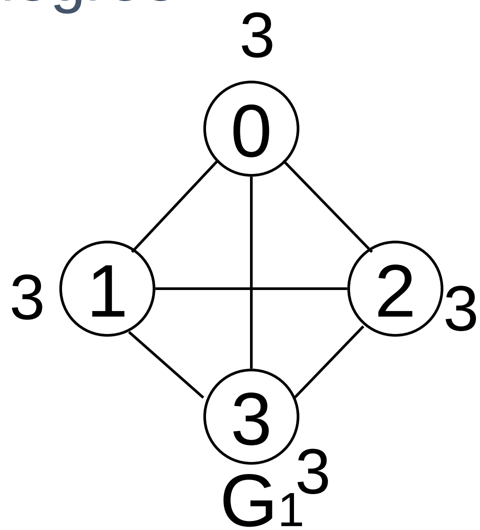


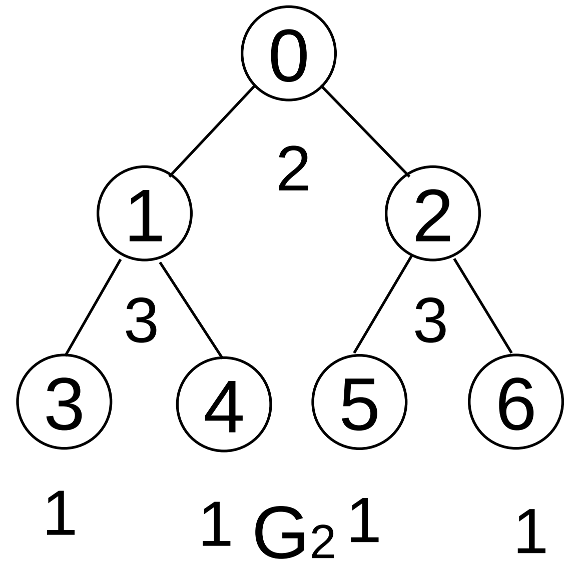
Degree

- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex v is the number of edges that have v as the head
  - the out-degree of a vertex v is the number of edges that have v as the tail

undirected graph

degree

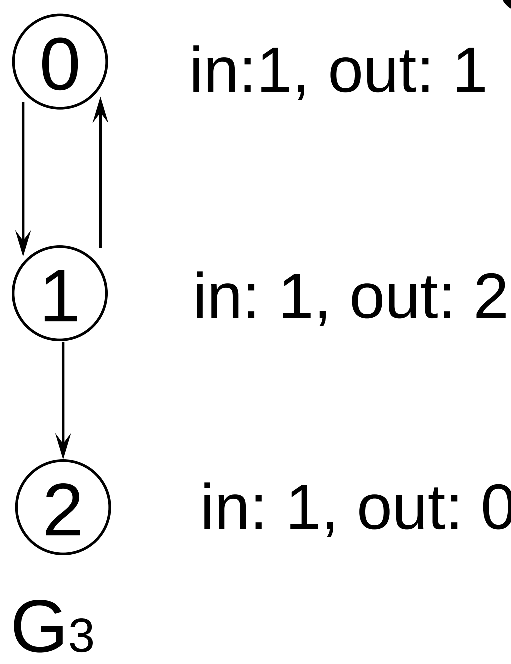




directed graph

in-degree

out-degree



ADT for Graph

functions: for all *graph* $\in$ *Graph*, v, v_1 and $v_2 \in$ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, v)::= return a graph with v inserted. v has no incident edge.

Graph InsertEdge(*graph*, v_1,v_2)::= return a graph with new edge between v_1 and v_2

Graph DeleteVertex(*graph*, v)::= return a graph in which v and all edges incident to it are removed

Graph DeleteEdge(*graph*, v_1, v_2)::=return a graph in which the edge (v_1, v_2) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE

else return FALSE

List Adjacent(*graph*,v)::= return a list of all vertices that are adjacent to v

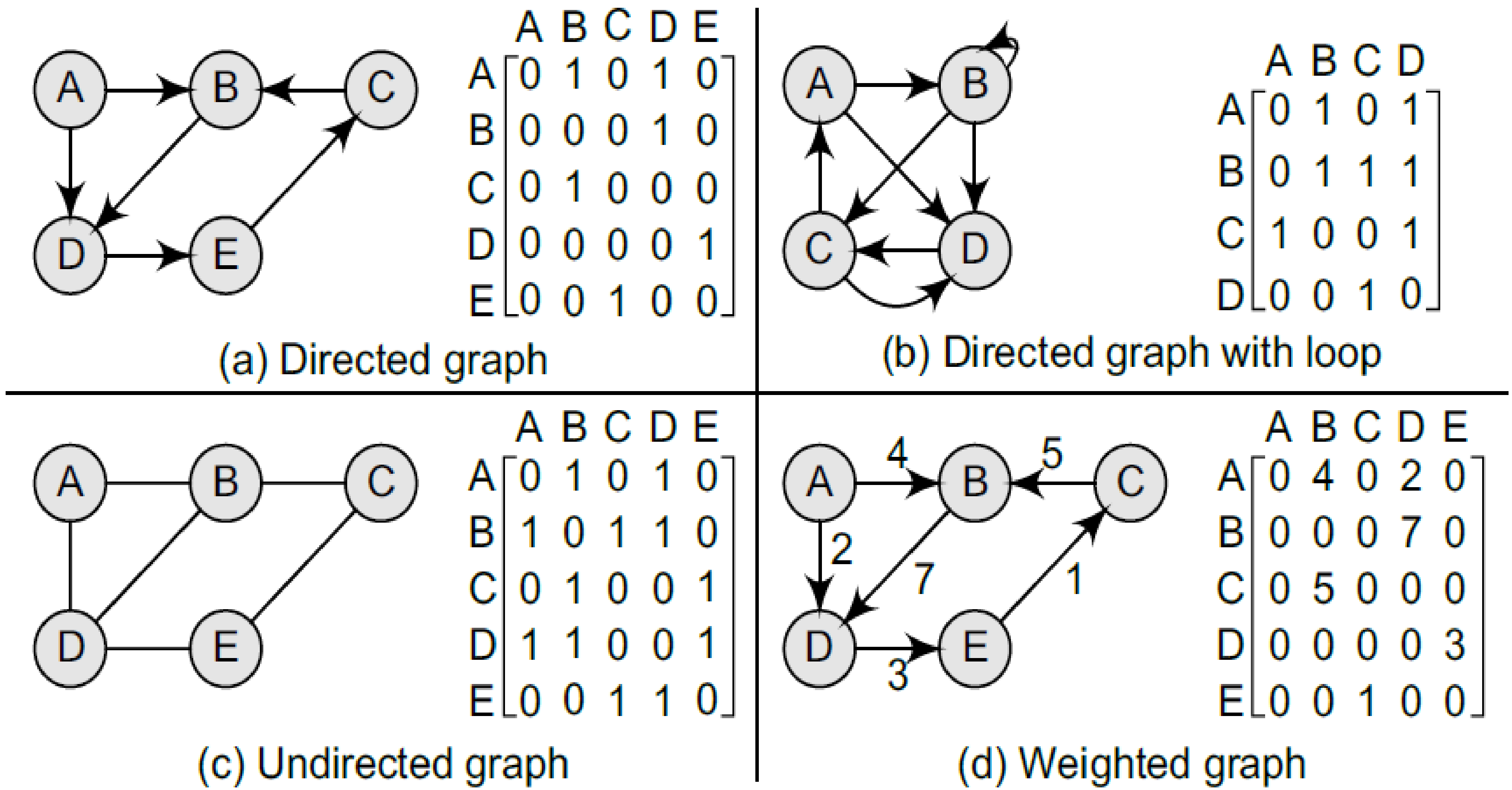
Graph Representations

- Adjacency Matrix
- Adjacency Lists

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Example


A B C D E
A 0 1 0 1 0
B 0 0 0 1 0
C 0 1 0 0 0
D 0 0 0 0 1
E 0 0 1 0 0
(a) Directed graph
A B C D
A 0 1 0 1
B 0 1 1 1
C 1 0 0 1
D 0 0 1 0
(b) Directed graph with loop
A B C D E
A 0 1 0 1 0
B 1 0 1 1 0
C 0 1 0 0 1
D 1 1 0 0 1
E 0 0 1 1 0
(c) Undirected graph
4
5
2
7
1
3
A B C D E
A 0 4 0 2 0
B 0 0 0 7 0
C 0 5 0 0 0
D 0 0 0 0 3
E 0 0 1 0 0
(d) Weighted graph

Data Structures for Adjacency Lists

Each row in adjacency matrix is represented as an adjacency list.

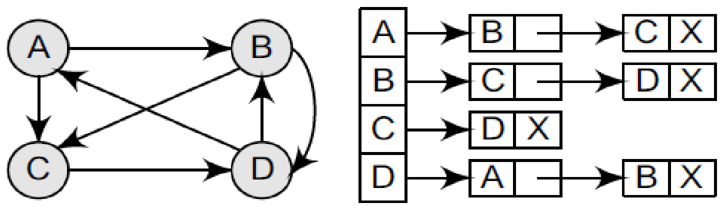
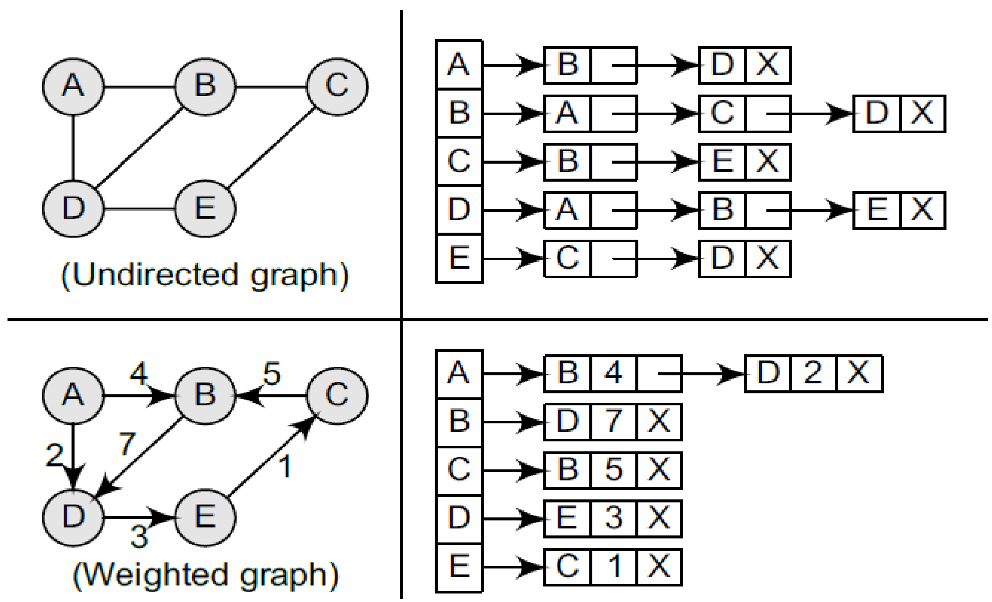
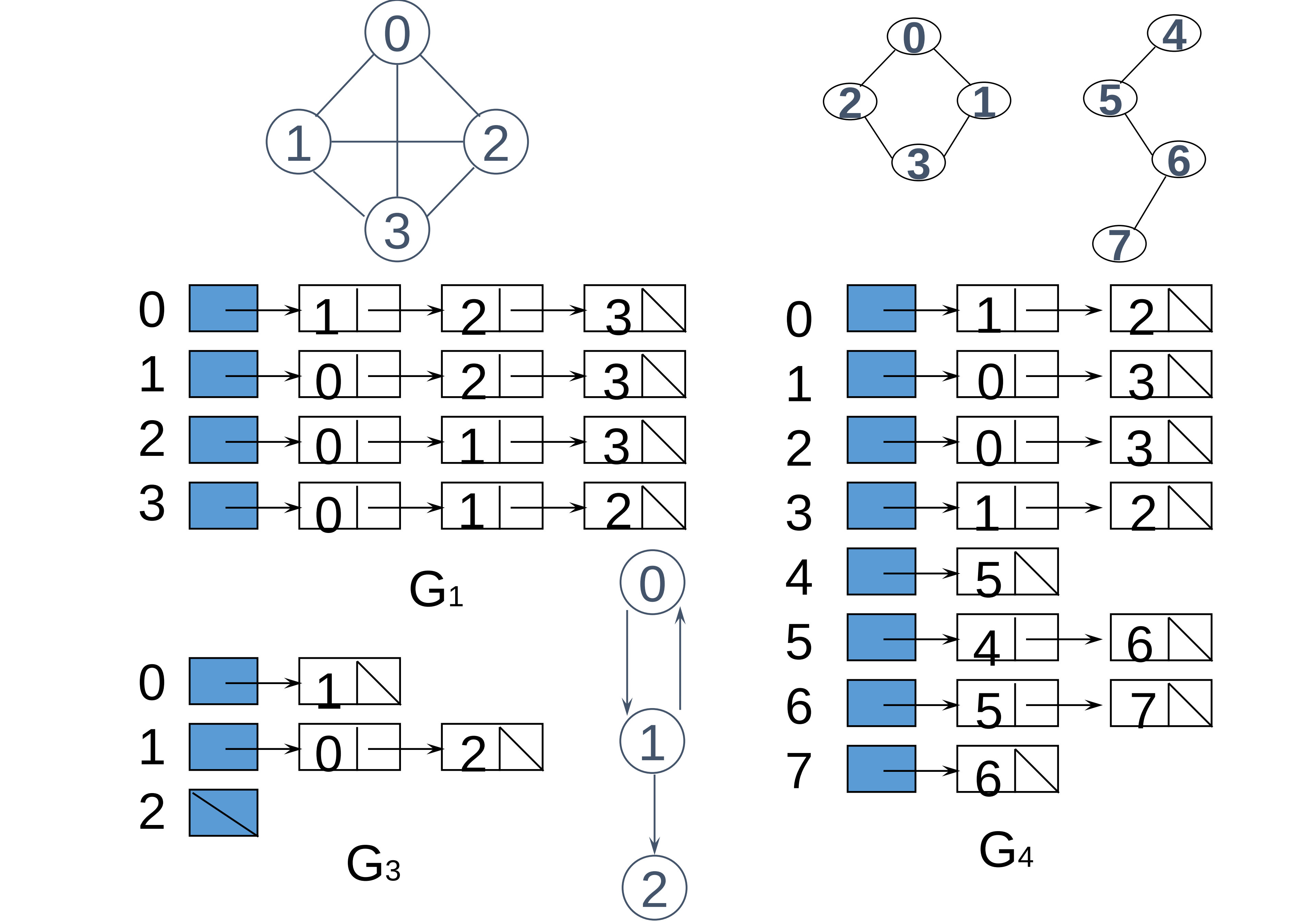


Figure 13.17 Graph G and its adjacency list





An undirected graph with n vertices and e edges ==> n head nodes and 2e list no

Graph Traversal

- Traversing a graph, we mean the method of examining the nodes and edges of the graph.
- There are two standard methods of graph traversal these two methods are:
- *Methods*: Depth-First-Search (DFS) or Breadth-First-Search (BFS)
- *Problem*: find a path between two nodes of the graph (e.g., Austin and Washington)

Depth First Search (DFS)

- The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.
- When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- Travel as far as you can down a path
- Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS can be implemented efficiently using a *stack*

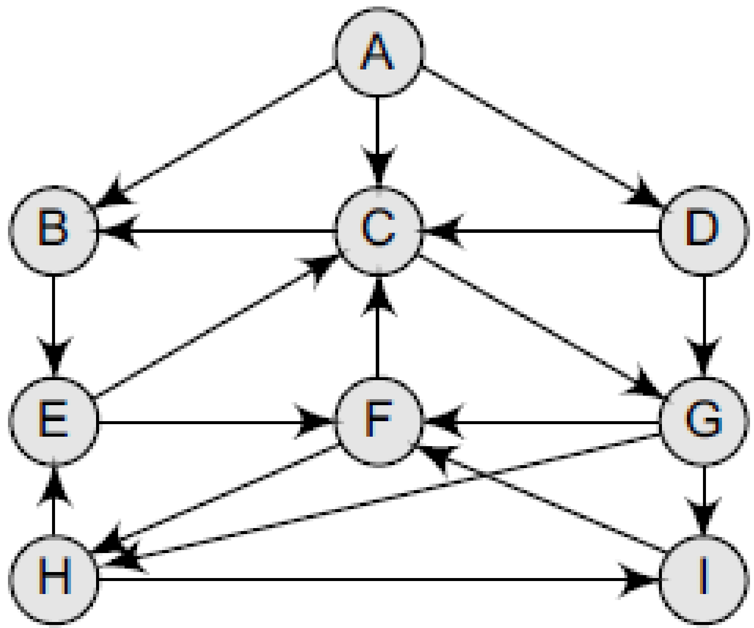
Value of status and significance

Status	State of the node	Description
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

DFS Algorithm:

- Step 1: SET STATUS = 1 (ready state) for each node in G
- Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- Step 3: Repeat Steps 4 and 5 until STACK is empty
- Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
- Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
- [END OF LOOP]
- Step 6: EXIT

Example of DFS



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Solution:

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H   STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I   STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F   STACK: E, C

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C   STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G   STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B   STACK: E

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E   STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

Applications of Depth-First Search Algorithm

- Depth-first search is useful for:
- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

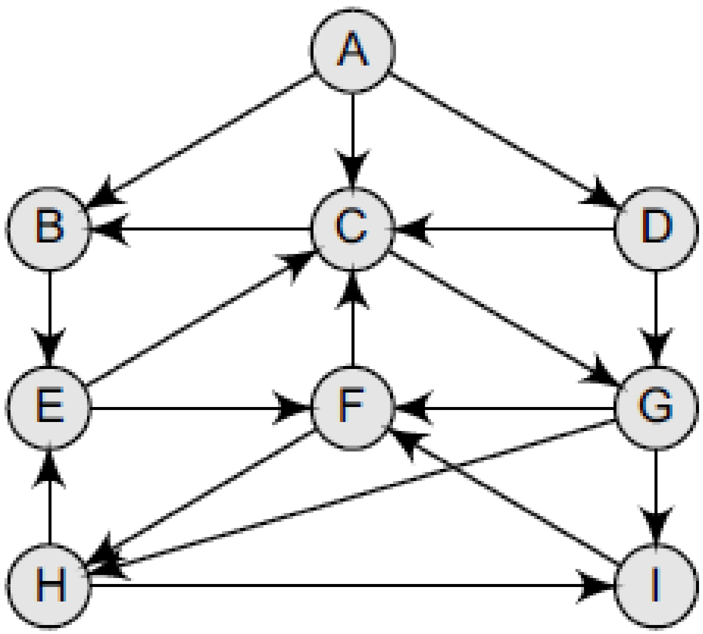
Breadth-First-Searching (BFS)

- Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes.
- Then for each of those nearest nodes, BFS explores their unexplored neighbour nodes, and so on, until it finds the goal.
- Look at all possible paths at the same depth before you go at a deeper level
- Back up as far as possible when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)

BFS Algorithm:

- Step 1: SET STATUS = 1 (ready state) for each node in G
- Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
- Step 3: Repeat Steps 4 and 5 until QUEUE is empty
- Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
- Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
- [END OF LOOP]
- Step 6: EXIT

Example:



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Solution:

(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0	QUEUE = A
REAR = 0	ORIG = \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1	QUEUE = A	B	C	D
REAR = 3	ORIG = \0	A	A	A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2	QUEUE = A	B	C	D	E
REAR = 4	ORIG = \0	A	A	A	B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(e) Dequeue a node by setting `FRONT = FRONT + 1` and enqueue the neighbours of `D`. Also, add `D` as the `ORIG` of its neighbours. Note that `D` has two neighbours `C` and `G`. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(f) Dequeue a node by setting `FRONT = FRONT + 1` and enqueue the neighbours of `E`. Also, add `E` as the `ORIG` of its neighbours. Note that `E` has two neighbours `C` and `F`. Since `C` has already been added to the queue and it is not in the Ready state, we will not add `C` and add only `F`.

FRONT = 5	QUEUE = A	B	C	D	E	G	F
REAR = 6	ORIG = \0	A	A	A	B	C	E

(g) Dequeue a node by setting `FRONT = FRONT + 1` and enqueue the neighbours of `G`. Also, add `G` as the `ORIG` of its neighbours. Note that `G` has three neighbours `F`, `H`, and `I`.

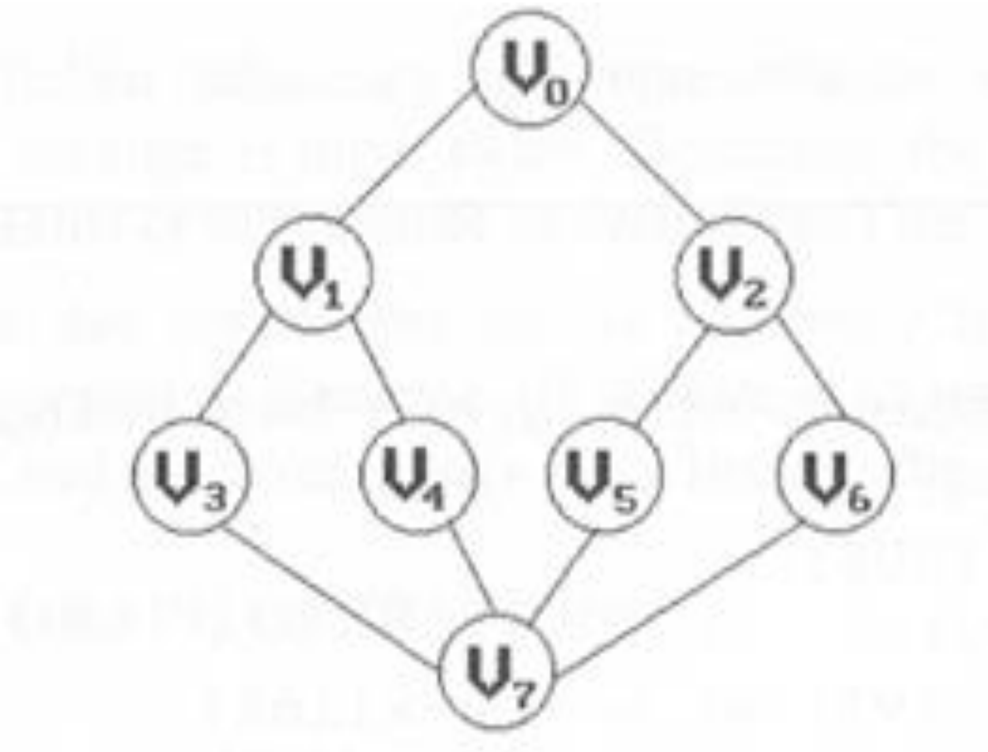
FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 9	ORIG = \0	A	A	A	B	C	E	G	G

Since `F` has already been added to the queue, we will only add `H` and `I`. As `I` is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the `QUEUE`. Now, backtrack from `I` using `ORIG` to find the minimum path `P`. Thus, we have `P` as `A -> C -> G -> I`.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.



Depth First Search: **v0, v1, v3, v7, v4, v5, v2, v6**

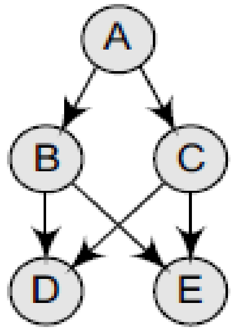
Breadth First Search: **v0, v1, v2, v3, v4, v5, v6, v7**

Topological Sorting

- Topological sorting is a way of arranging the nodes (or vertices) of a directed graph in a linear order such that for every directed edge from node A to node B, node A comes before node B in the ordering.
- In simpler terms, it's like making a list of tasks where some tasks depend on others. For example, if task A needs to be done before task B, topological sorting will arrange the tasks so that A comes before B. This is commonly used in scenarios like:
  - Scheduling jobs based on dependencies (like prerequisites in a course plan).
  - Determining the order of compilation in programming (certain files must be compiled before others).

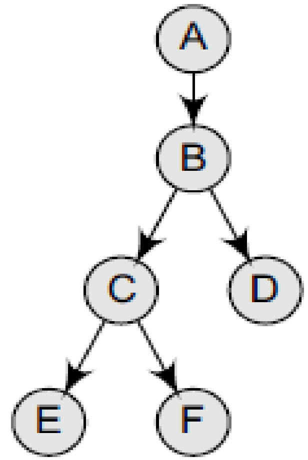
- Topological order: linear ordering of vertices of a graph
- A topological sort takes a directed acyclic graph and produces a linear ordering of all its vertices such that if the graph G contains an edge (v,w) then the vertex v comes before the vertex w in the ordering.

Topological Sort



Topological sort can be given as:

- A, B, C, D, E
- A, B, C, E, D
- A, C, B, D, E
- A, C, B, E, D

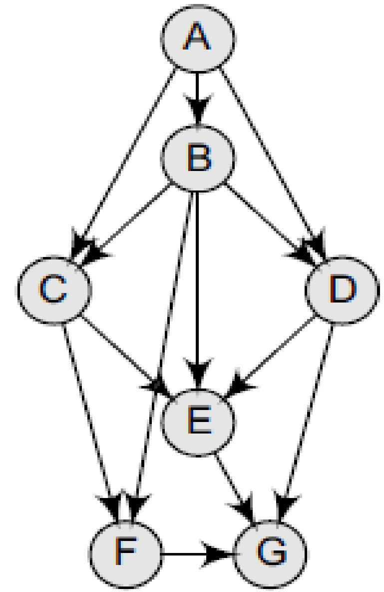


Topological sort can be given as:

- A, B, D, C, E, F
- A, B, D, C, F, E
- A, B, C, D, E, F
- A, B, C, D, F, E

....

- A, B, F, E, D



Topological sort can be given as:

- A, B, C, F, D, E, C
- A, B, C, D, E, F, G
- A, B, C, D, F, E, G
- A, B, D, C, E, F, G

....

- A, B, D, C, F, E, G

Algorithm

1. Compute the indegrees of all vertices
2. Find a vertex U with indegree 0 and print it (store it in the ordering)

   If there is no such vertex then there is a cycle and the vertices cannot be ordered. Stop.
3. Remove U and all its edges (U,V) from the graph.
4. Update the indegrees of the remaining vertices.
5. Repeat steps 2 through 4 while there are vertices to be processed.