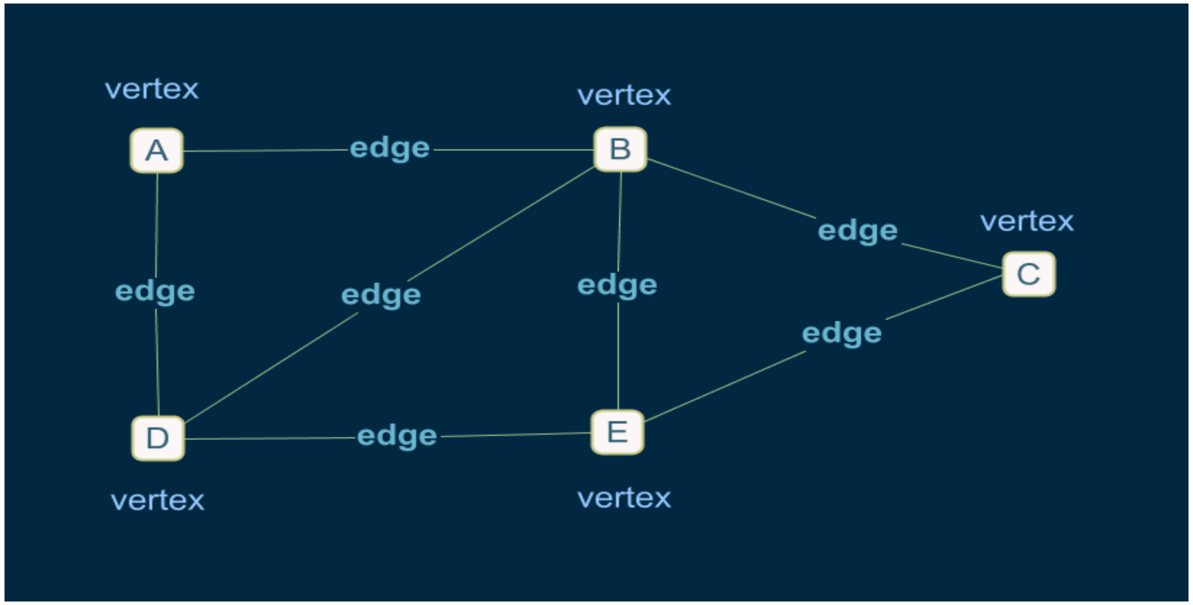
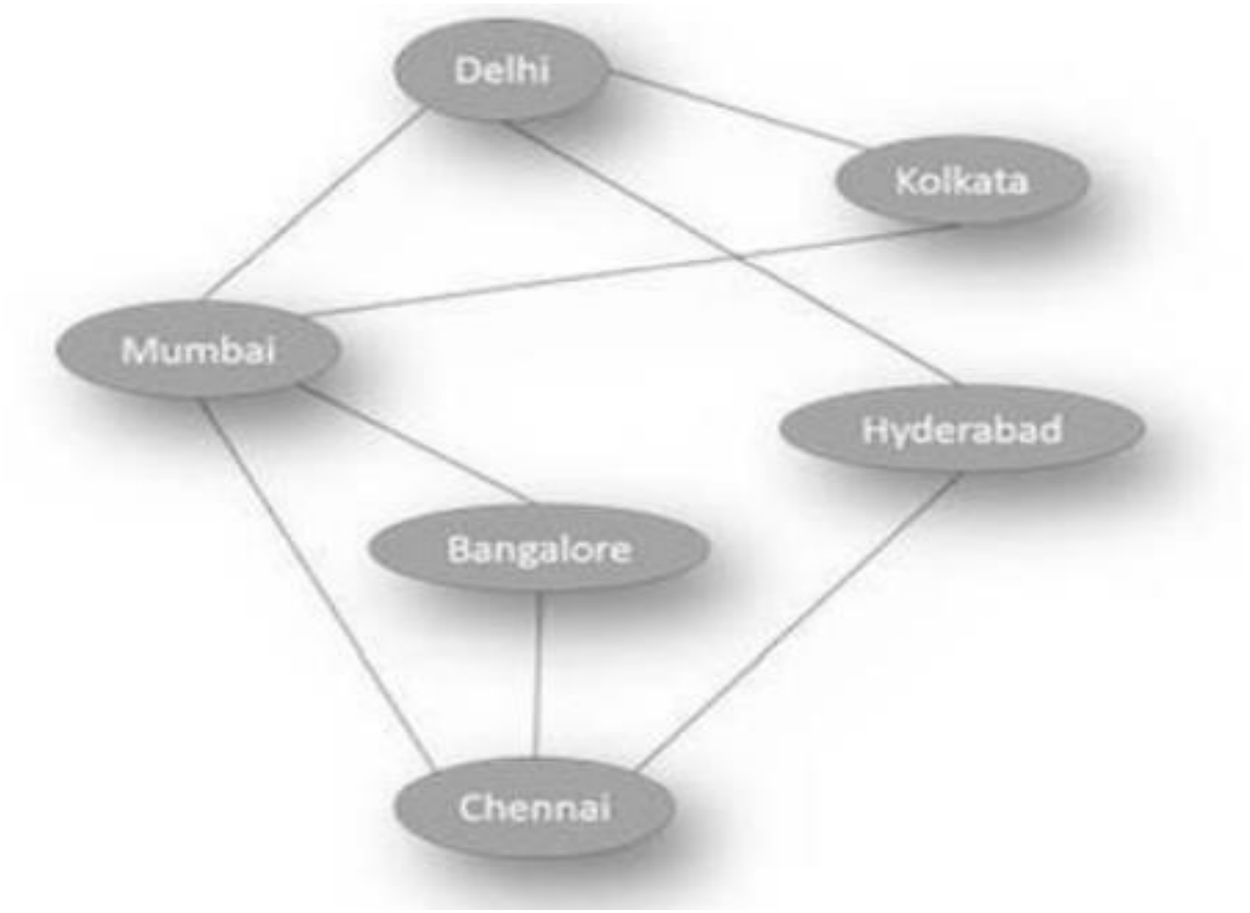
Graphs

What is a graph?

A Graph is represented by G where G = (V, E), where V is a finite set of points called Vertices and E is a finite set of Edges.

Each edge is a connecting line of two vertices and is denoted by a pair (u, v) where u, v belongs to set of vertices V.





A graph can be of 2 types

Formal definition of graphs

A graph *G* is defined as follows:

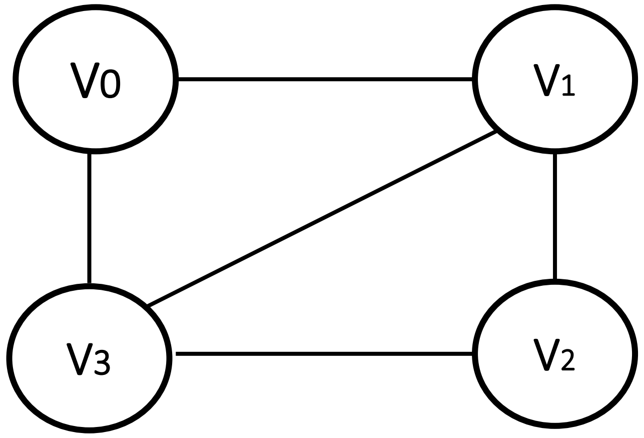
$$G=(V,E)$$

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

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

Undirected graphs

When the edges in a graph have no direction, the graph is called *undirected.* The graph below has 4 vertices and 5 edges

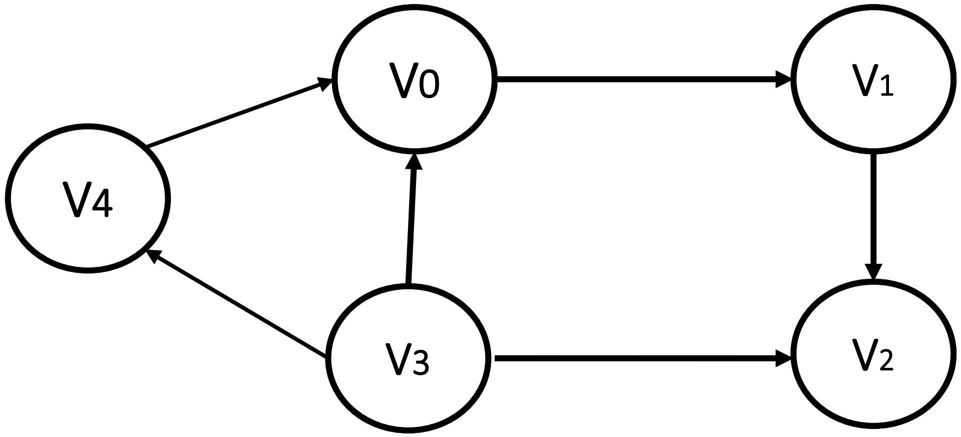


$V(G) = \{ V_0, V_1, V_2, V_3 \}$

$E(G) = \{ (V_0, V_1), (V_0, V_3), (V_1, V_2), (V_1, V_3), (V_2, V_3) \}$

Directed graphs

When the edges in a graph have a direction, the graph is called *directed* (or *digraph*). The graph below has 5 vertices and 6 edges

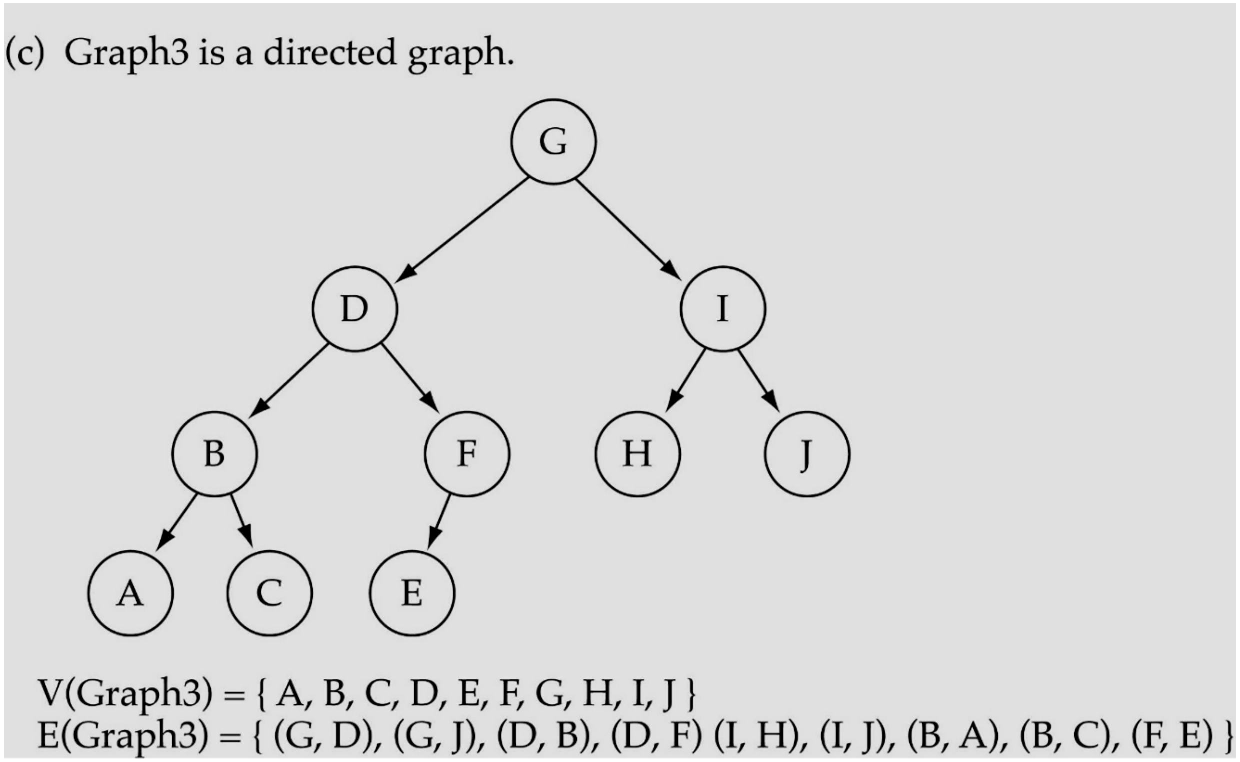


$V(G) = \{ V_0, V_1, V_2, V_3, V_4 \}$

$E(G) = \{ (V_0, V_1), (V_1, V_2), (V_3, V_2), (V_3, V_0), (V_3, V_4), (V_4, V_0), \}$

Trees vs graphs

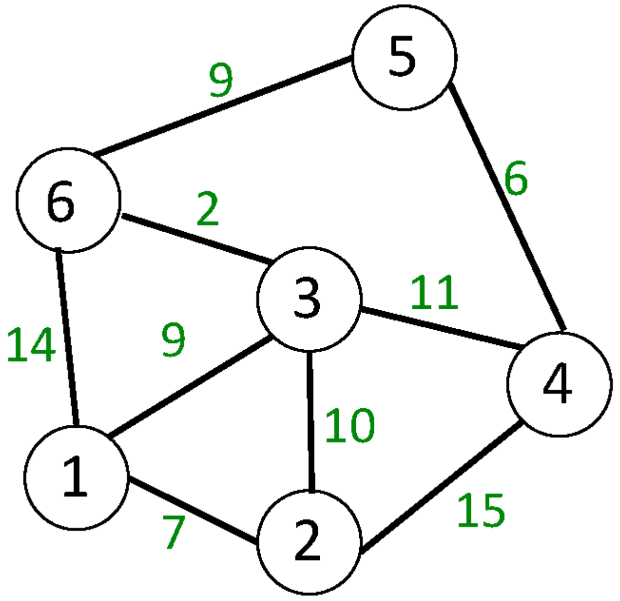
Trees are special cases of graphs!!

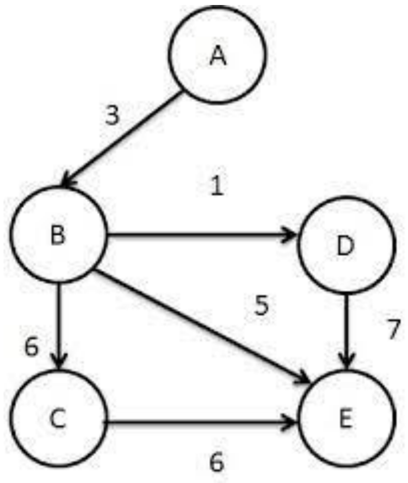

(c) Graph3 is a directed graph.

V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, J), (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

Graph terminology

Weighted Graph: A graph is weighted if its edges have been assigned a non negative value as weight. The weight on the edge may represent cost, length or distance associated with the edge.





Graph terminology

Subgraph : A graph H is said to be subgraph of another graph G, if the vertex set of H is subset of vertex set of G and edge set is H is subset of edge set of G

Adjacency: Adjacency is a relation between two vertices of a graph. A vertex v is adjacent to another vertex u if there is an edge from vertex u to vertex v.

In an undirected graph, if we have edge (u,v), then u is adjacent to v and v is adjacent to u. So adjacency relation is symmetric in undirected graph.

Graph terminology

In directed graph, if we have edge (u,v), then u is adjacent to v, and v is adjacent from u.

Incidence: Incidence is a relation between vertex and an edge of a graph.

In an undirected graph the edge (u,v) is incident on vertices u and v.

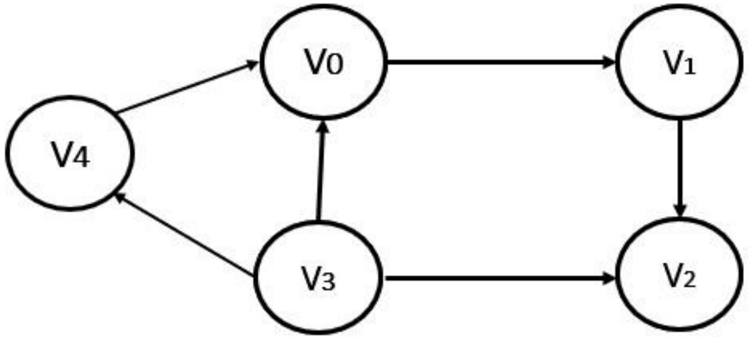
In directed graph the edge (u,v) is incident from vertex u and incident to vertex v.

Graph terminology

Path: A Path is a sequence of edges between two vertices.
In graph below,
V3 -> V4 -> V0 -> V1 -> V2 is a path

Length of a Path: The length of a Path is the total number of edges included in the path.

Reachable: If there is Path P from vertex u to vertex v, then vertex v is said to be reachable from vertex u via path P. For e.g V2 is reachable from V3, but V3 is not reachable from V4

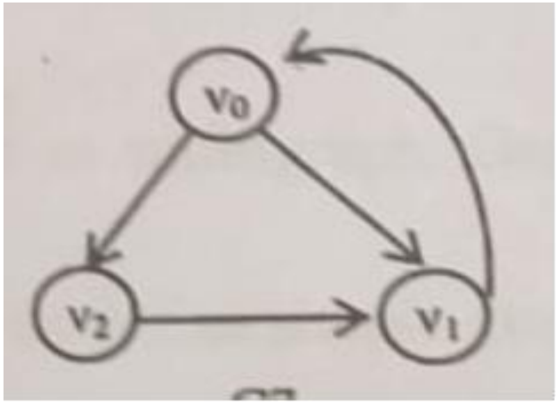


Graph terminology

Simple Path: A path in which all vertices are distinct

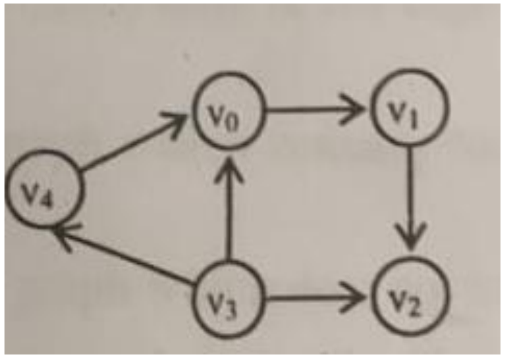
Cycle: A Cycle is a path that starts and ends at the same vertex and include at least one vertex. There has to be at least 3 vertices to form a cycle in undirected graph

Cyclic graph: A graph that has one or more cycles is called a cyclic graph.



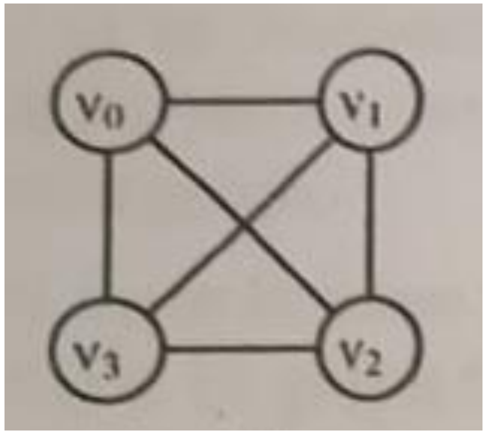
Graph terminology

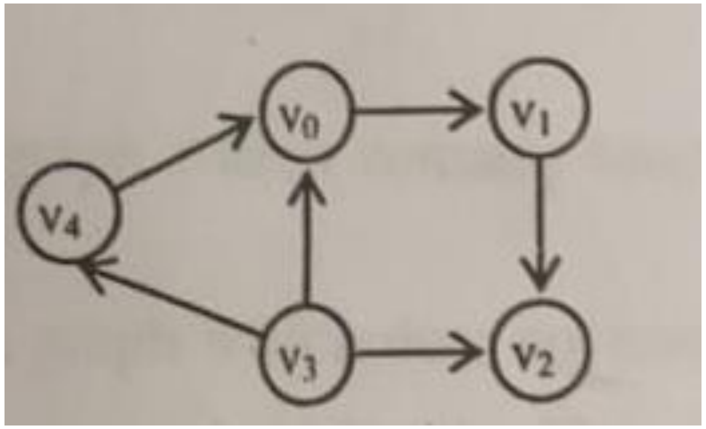
Acyclic graph: A graph that has no cycle is called Acyclic graph.



DAG: A directed acyclic graph is named as DAG.

Degree: In an undirected graph, the degree of a vertex is the number of edges incident on it.



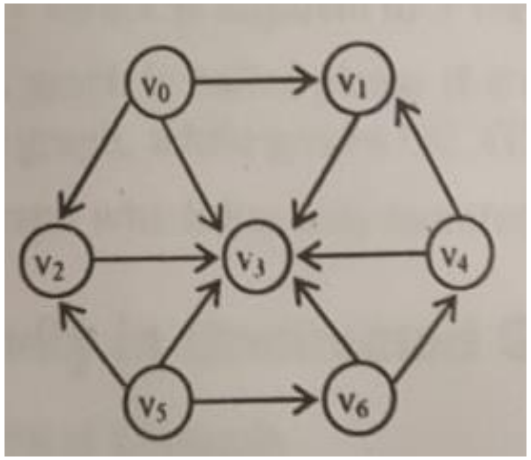


In directed graph, each vertex has an indegree and an outdegree. The degree of each vertex in directed graph is sum of its indegree and outdegree.

Indegree of a vertex is number of edges entering into vertex
Outdegree of a vertex is number of edges leaving the vertex.

Graph terminology

Source: A vertex, which has no incoming edge, but has outgoing edges is called as a source



Vertex V0 and V5 are source

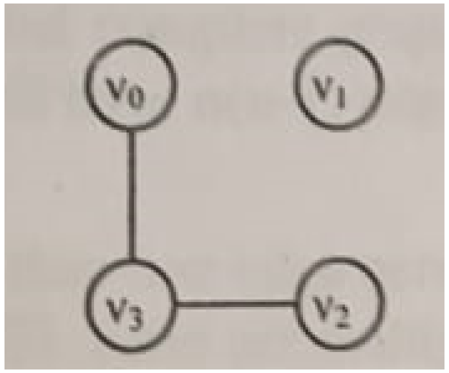
Sink: A vertex, which has no outgoing edge but has incoming edges is called as a sink

Vertex V3 is sink

Graph terminology

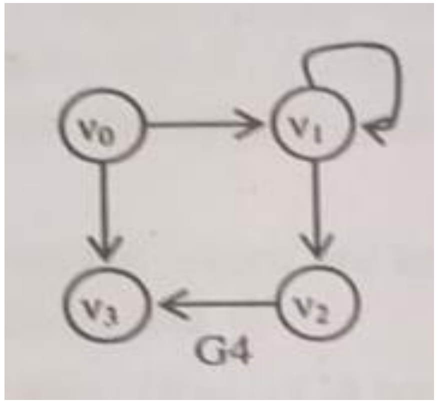
Pendent vertex: A vertex in a digraph is pendent if its indegree is 1 and outdegree is 0

Isolated vertex: If the degree of a vertex is 0, then it is called as Isolated vertex



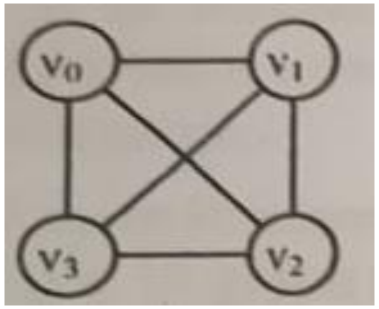
V1 is isolated

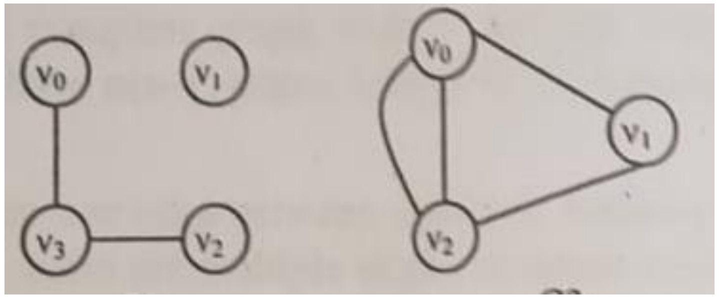
Loop: A self loop is an edge that connects a vertex to itself.



Loop at V1

Planar graph: A graph which can drawn on plane paper without any two edges intersecting each other.



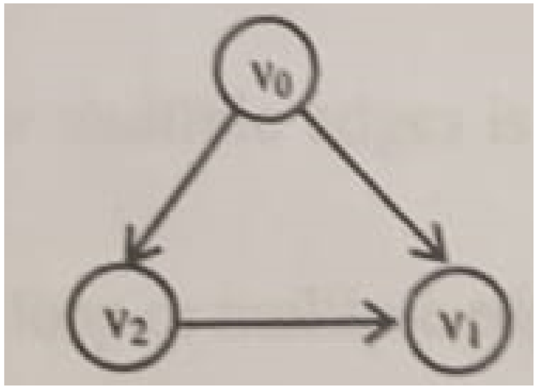


Not a planar

planar

Graph terminology

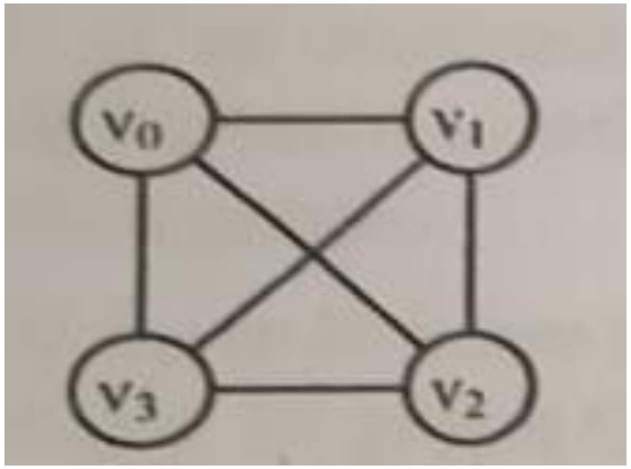
Successor and predecessor:



V0 is predecessor of V1 and V1 is successor of V0

Graph terminology

Regular graph: A graph in which each vertex is adjacent to the same number of vertices.



Every vertex is adjacent to 3 vertices

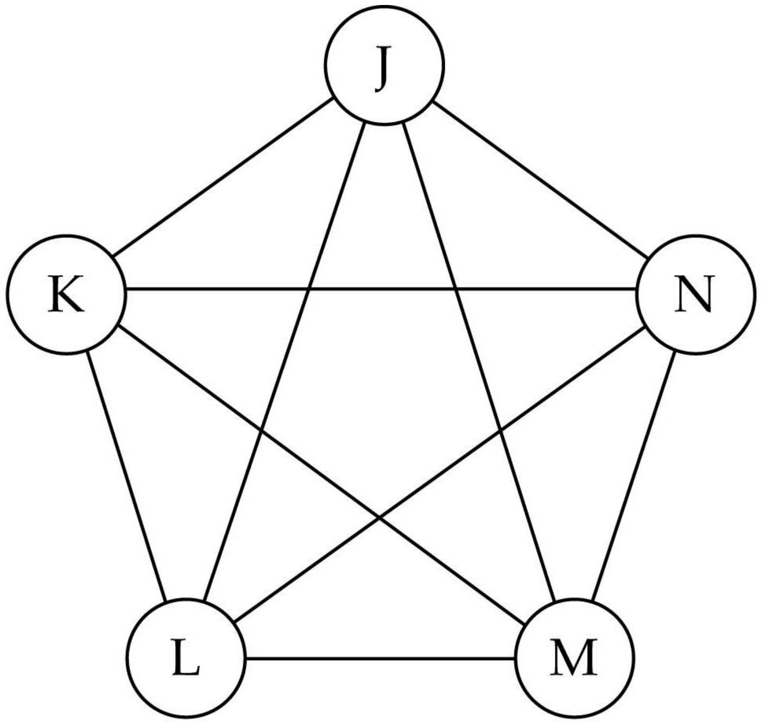
Null graph: A graph which has only isolated vertices is called Null graph.

Graph terminology

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

*N * (N-1) / 2*

$O(N^2)$



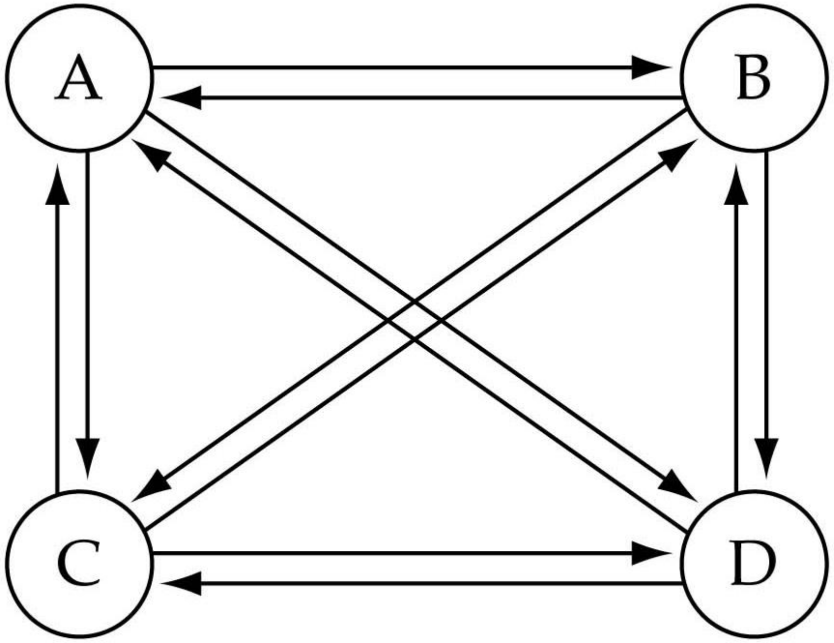
(b) Complete undirected graph.

Graph terminology (cont.)

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

  *N * (N-1)*

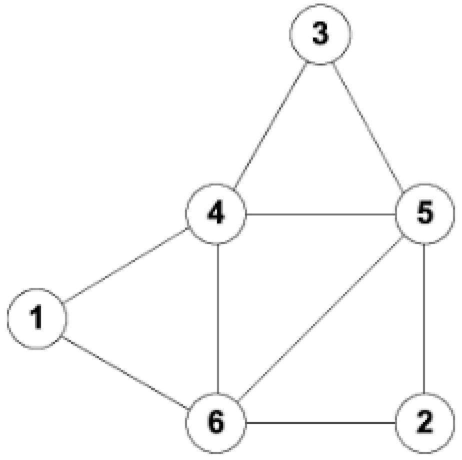
$O(N^2)$

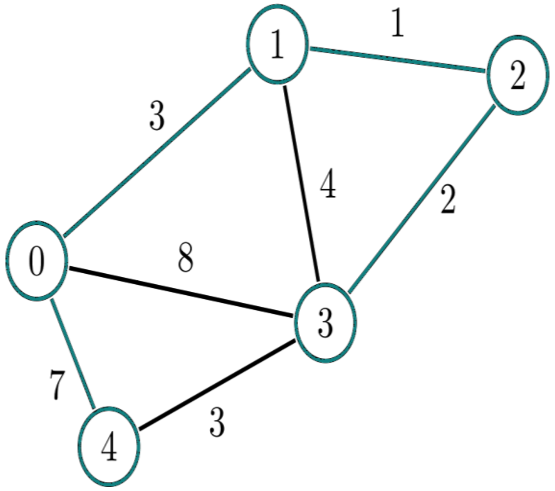


(a) Complete directed graph.

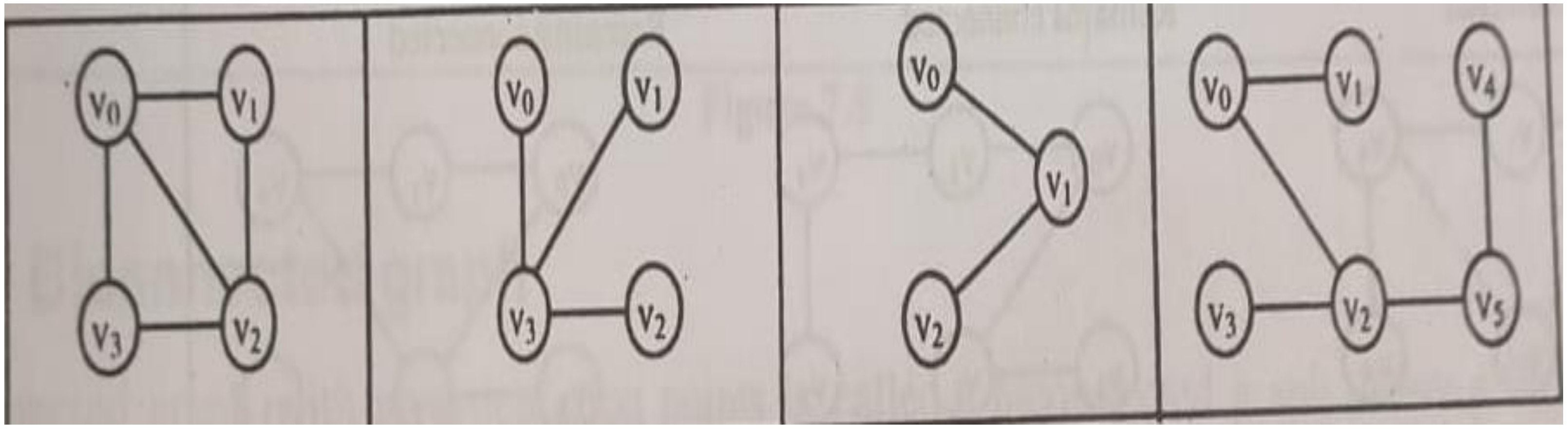
Graph terminology (cont.)

Connected graph: An undirected graph is said to be connected if there is a path from any vertex to any other vertex, or any vertex is reachable from any other vertex.



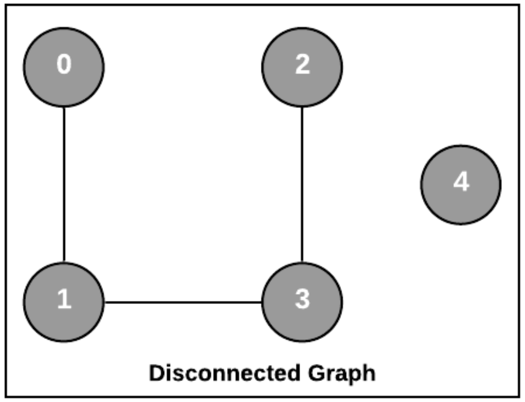


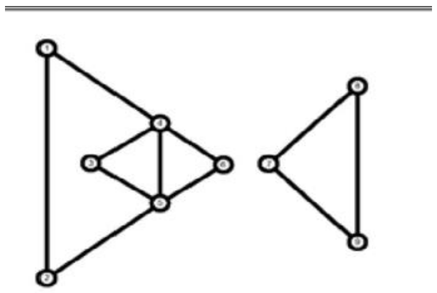
Graph terminology (cont.)

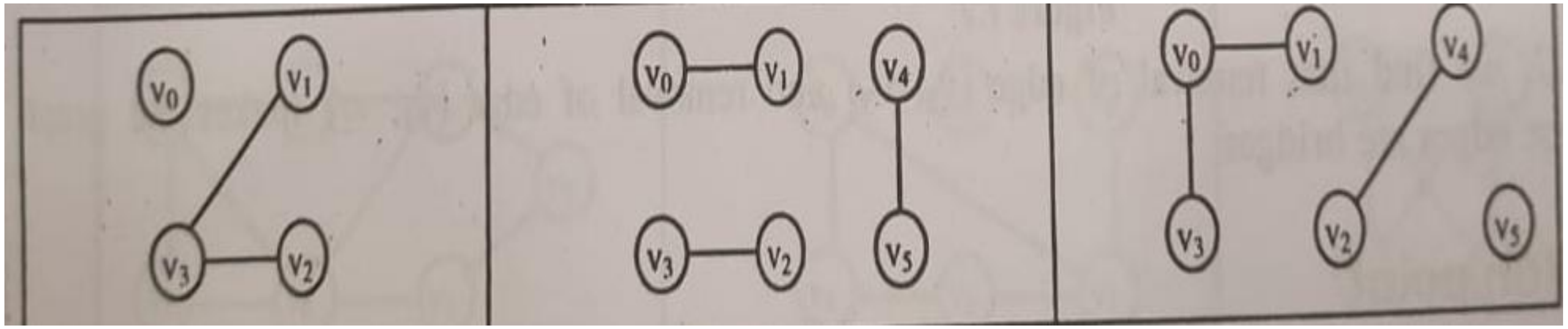


Graph terminology (cont.)

Disconnected undirected graph:

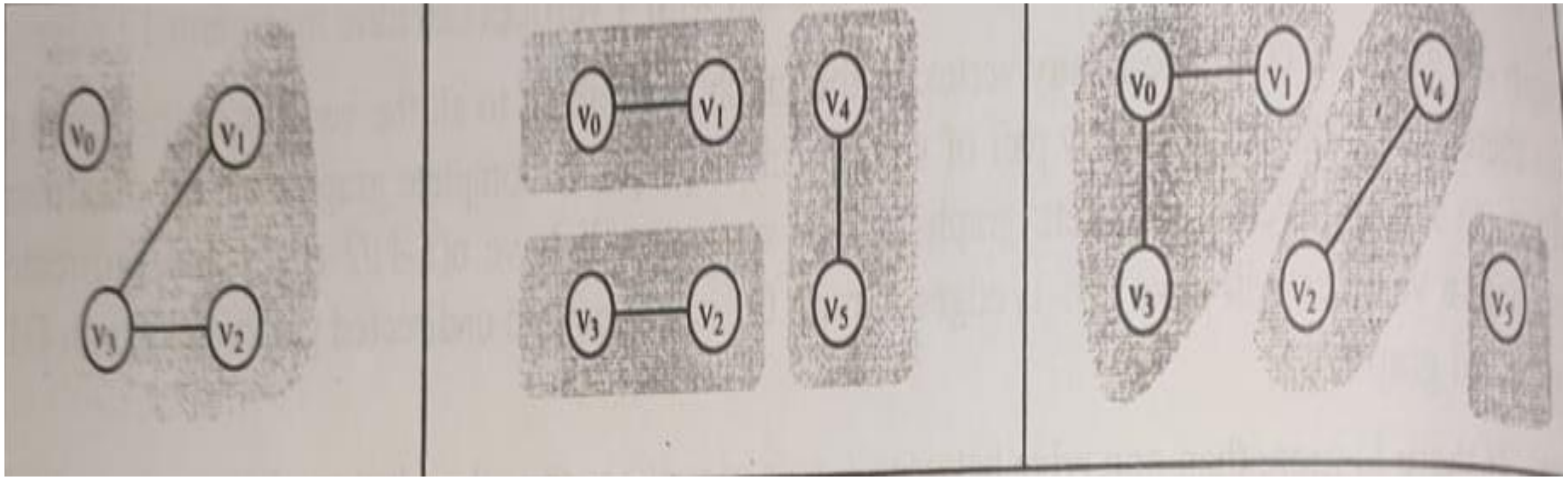




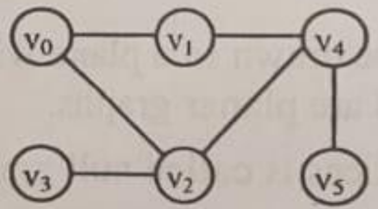


Graph terminology (cont.)

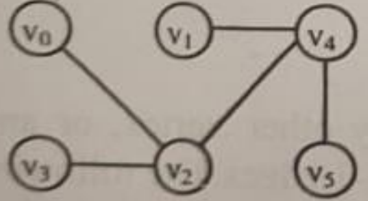
Connected components: An undirected graph which is not connected may have different parts of the graph which are connected. These parts are called as connected components.



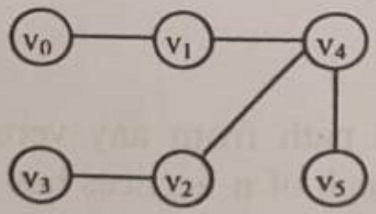
Bridge: If on removing an edge from a connected graph, the graph becomes disconnected then that edge is called as bridge. Consider graph below and remove edge one by one and identify bridge.



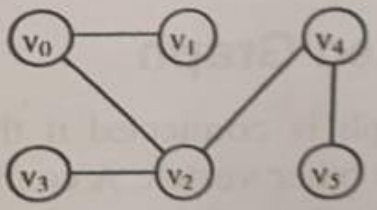
Connected Graph G



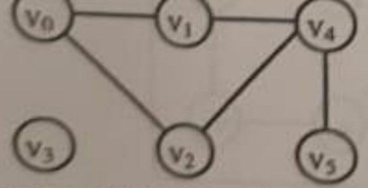
Remove Edge (v_0, v_1) from G
Remains connected



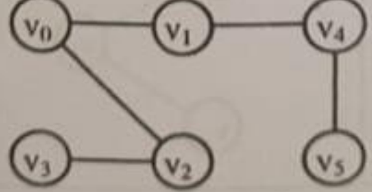
Remove Edge (v_0, v_2) from G
Remains connected



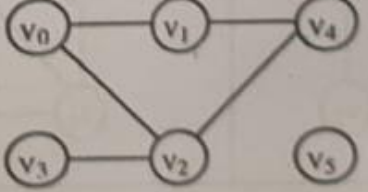
Remove Edge (v_1, v_4) from G
Remains connected



Remove Edge (v_2, v_3) from G
Becomes Disconnected

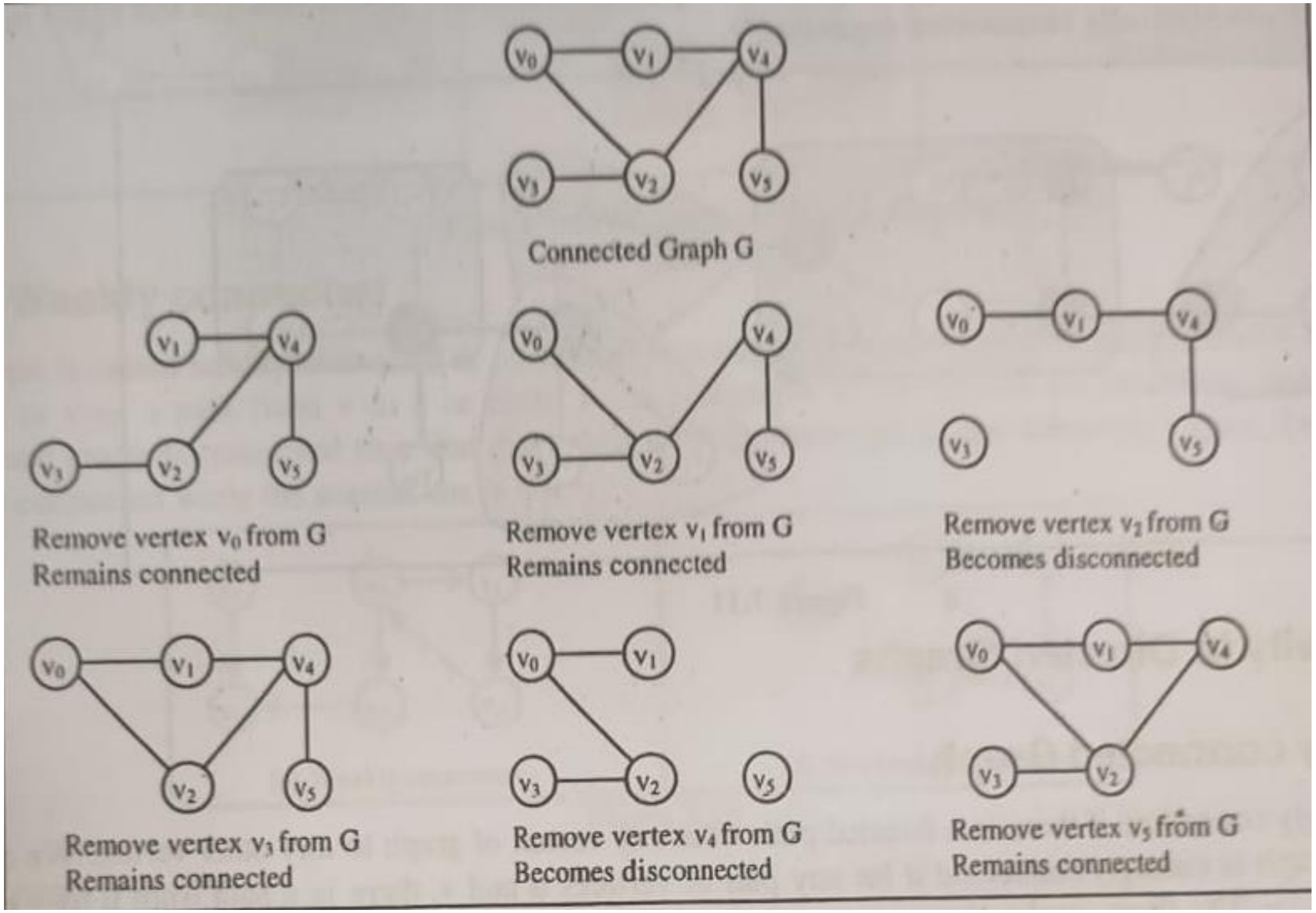


Remove Edge (v_2, v_4) from G
Remains connected



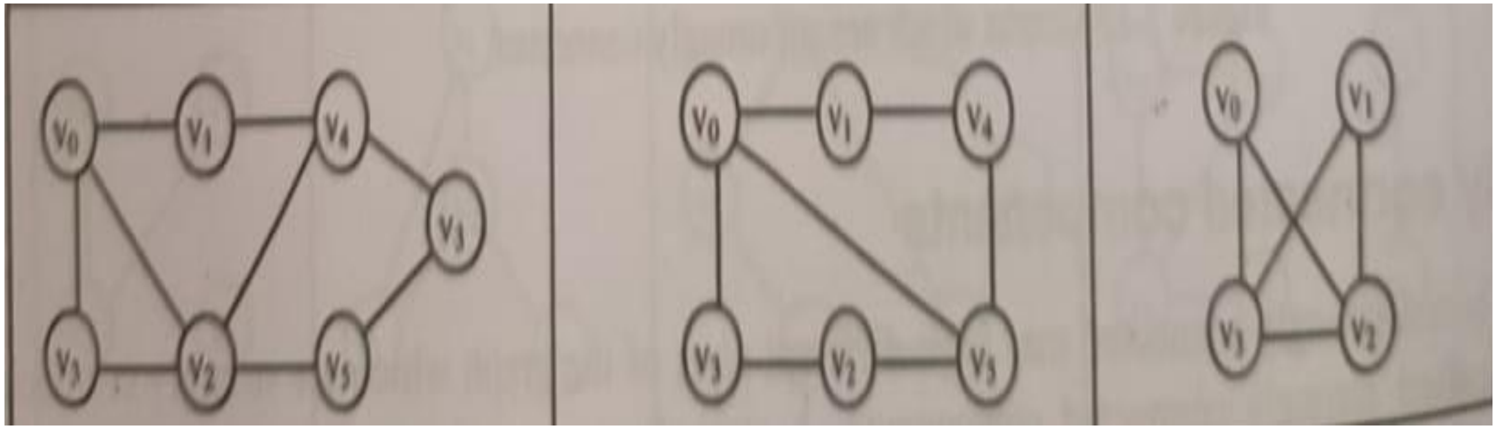
Remove Edge (v_4, v_5) from G
Becomes disconnected

Articulation point: If on removing a vertex from a connected graph, the graph becomes disconnected then that vertex is called the articulation point.

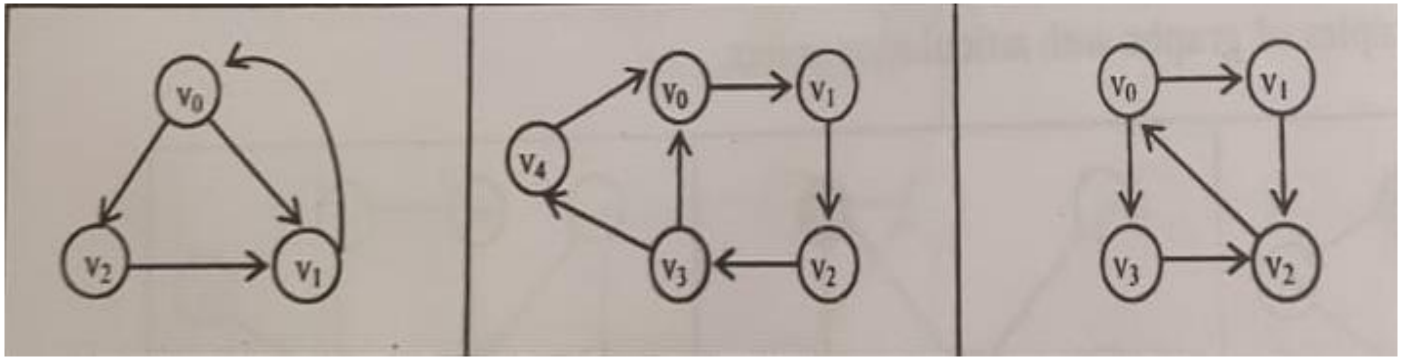


Graph terminology (cont.)

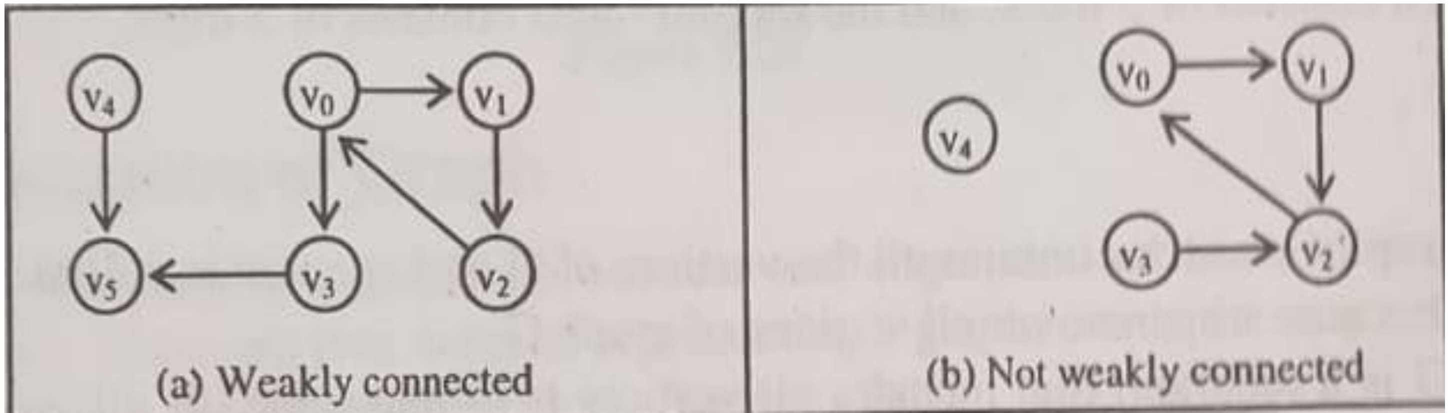
Biconnected graph: A connection graph with no articulation point is called a biconnected graph



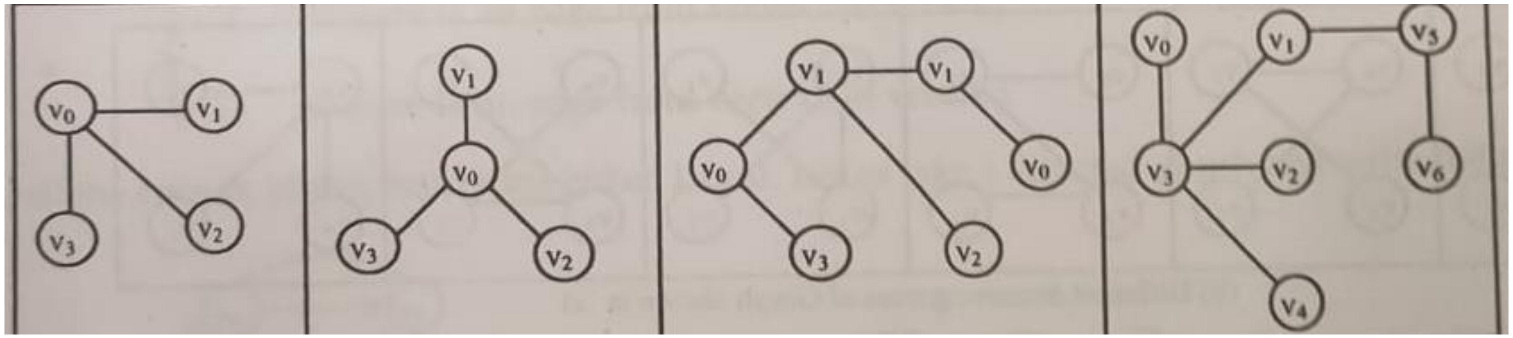
Strongly connected graph: A directed graph is strongly connected if there is a direct path from any vertex of graph to any other vertex.



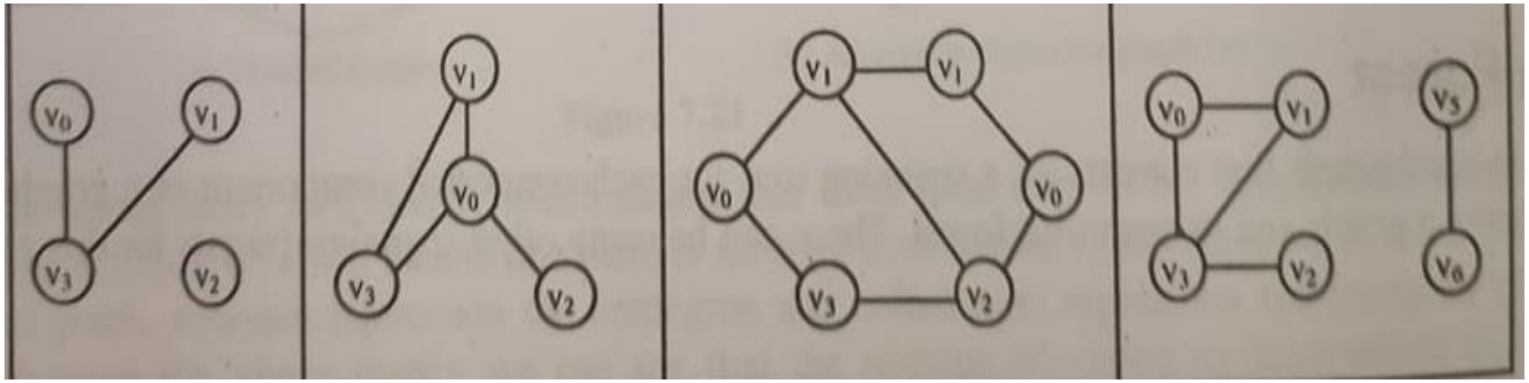
Weakly connected graph: A directed graph is weakly connected if for any pair of vertices u and v, there is a path from u to v or a path from v to u or both.



Tree: An undirected connected graph is called as Tree if there are no cycles in it. A tree with n vertices will have exactly n-1 edges.

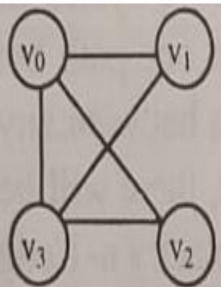


The following examples are graphs, which are not trees

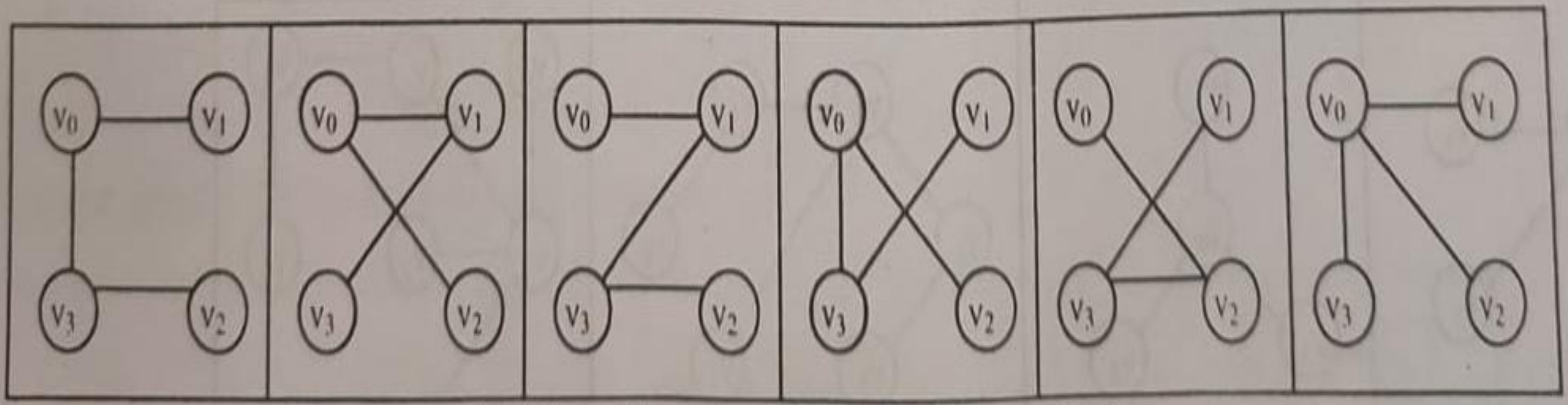


Spanning Tree: A subgraph T of a connected graph G, which contains all the vertices of G and is a tree is called a spanning tree of G.

Spanning tree of a graph is not unique, there can be more than one spanning trees of a graph.



(a) A Connected Graph



(b) Different Spanning trees of Graph shown in (a)

Applications of Graphs

- Representing relationships between components in electronic circuits
- Transportation networks: Highway network, Flight network
- Computer networks: Local area network, Internet, Web
- Databases: For representing ER (Entity Relationship) diagrams in databases, for representing dependency of tables in databases

Representation of Graph

We have mainly two parts in a graph, vertices and edges, and we have to design a Data Structure keeping in mind these parts. There are two ways of representing graph.

1. Sequential representation (Adjacency matrix)

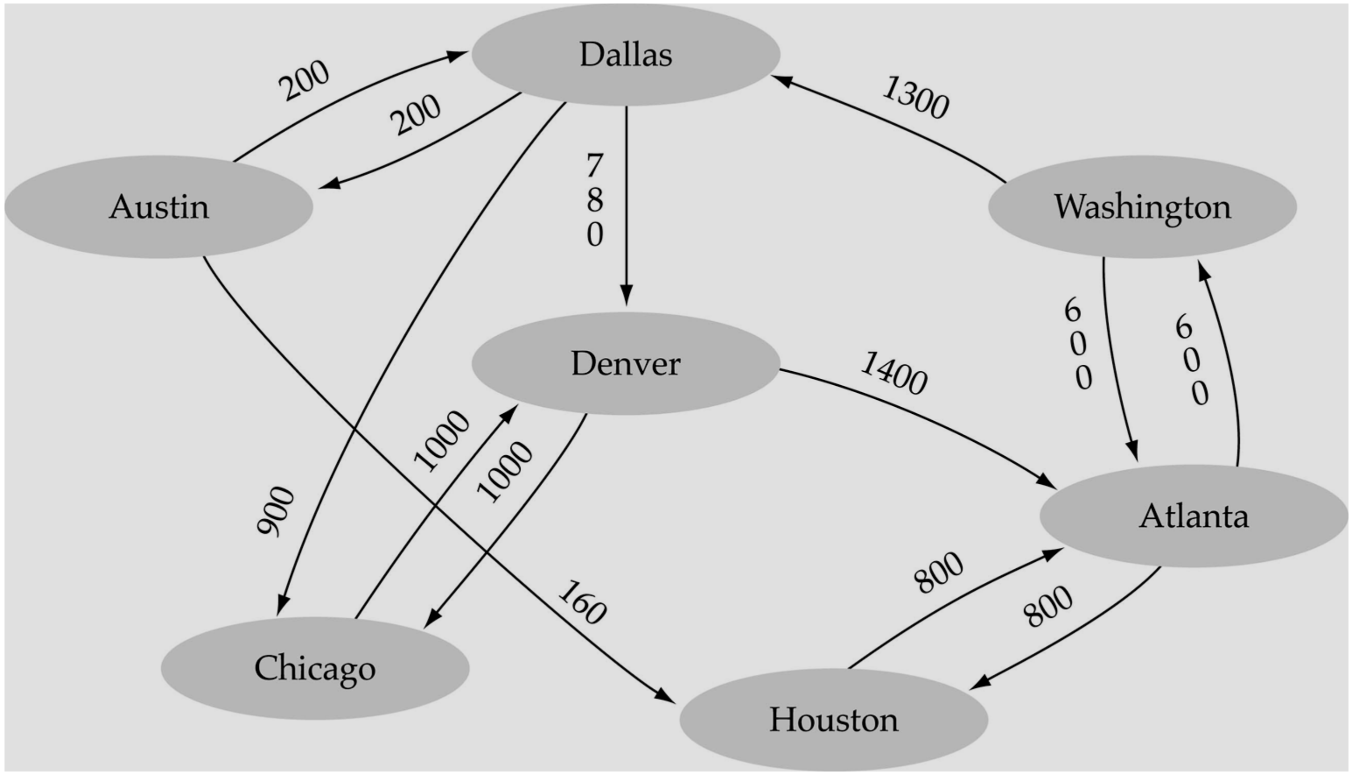
2. Linked representation (Adjacency list)

Adjacency Matrix

1. One of the ways to represent a graph is to use two-dimensional matrix.
2. Each combination of row and column represent a vertex in the graph.
3. The value stored at the location row v and column u is the edge from vertex v to vertex u.
4. The nodes that are connected by an edge are called adjacent nodes. This matrix is used to store adjacent relation so it is called the Adjacency Matrix.
5. In the below diagram, we have a graph and its Adjacency matrix.

Graph terminology (cont.)

<u>Weighted graph</u>: a graph in which each edge carries a value

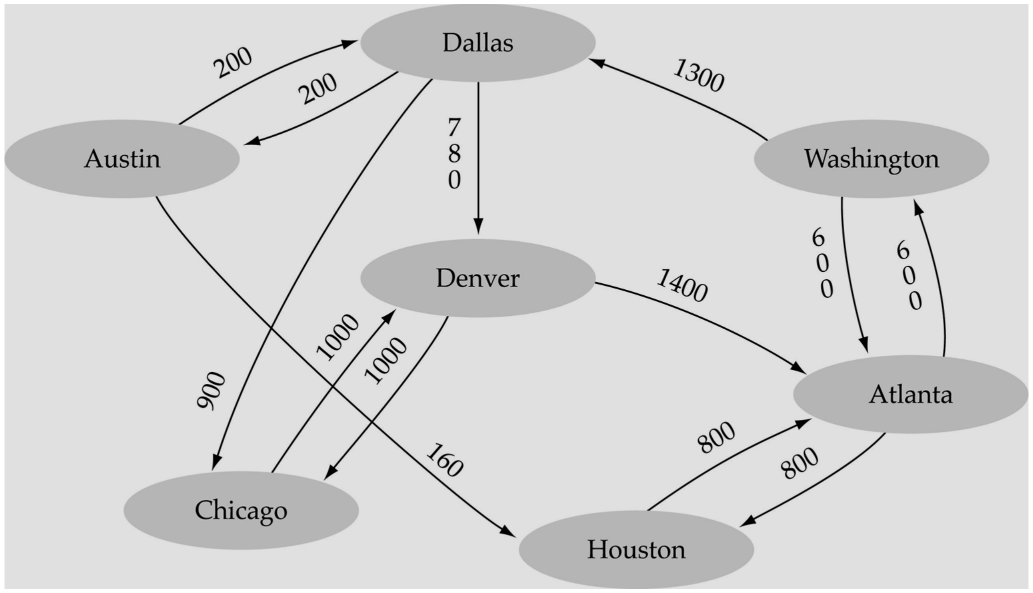


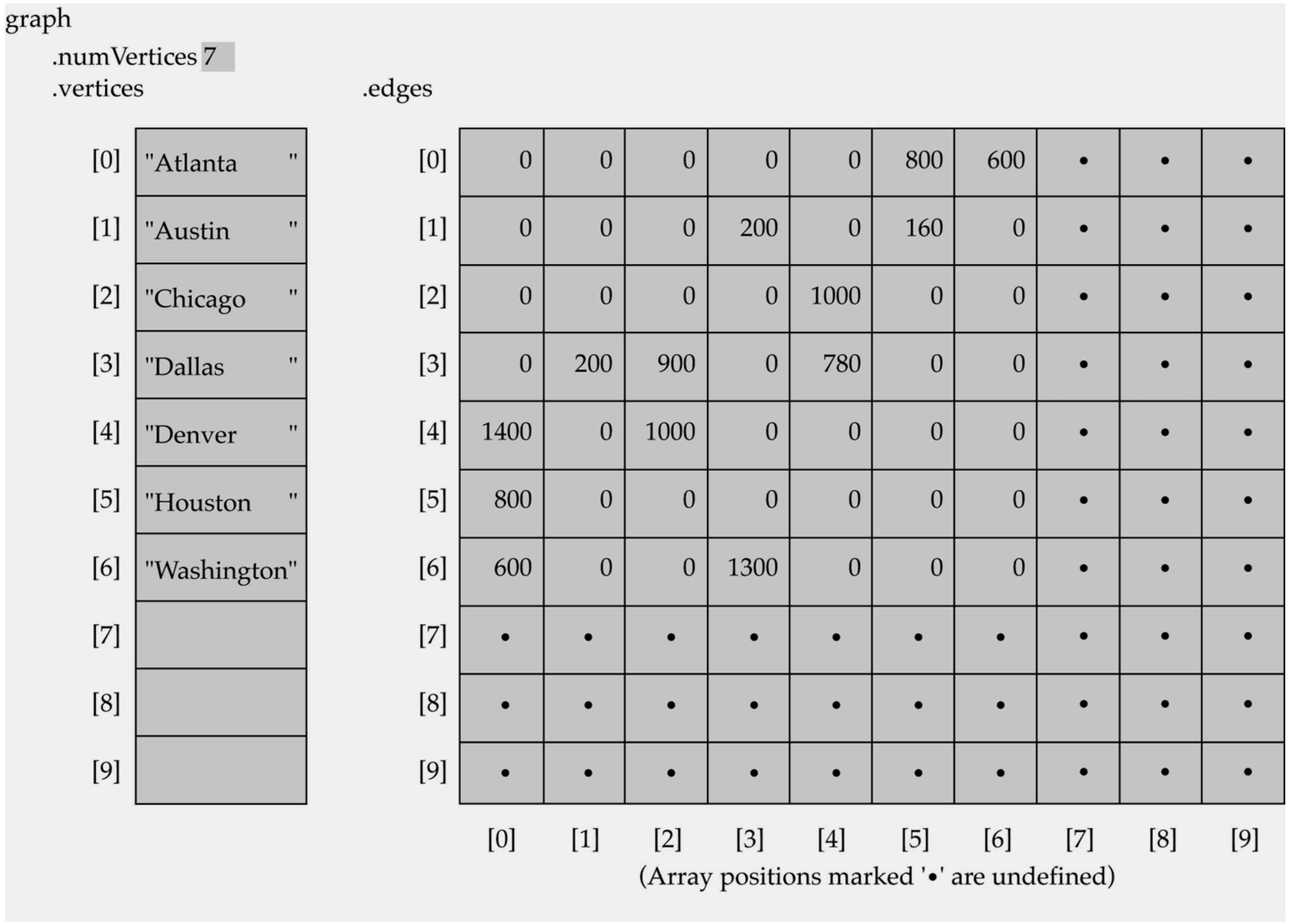
Graph implementation

Array-based implementation

A 1D array is used to represent the vertices

A 2D array (adjacency matrix) is used to represent the edges





graph

.numVertices 7

.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

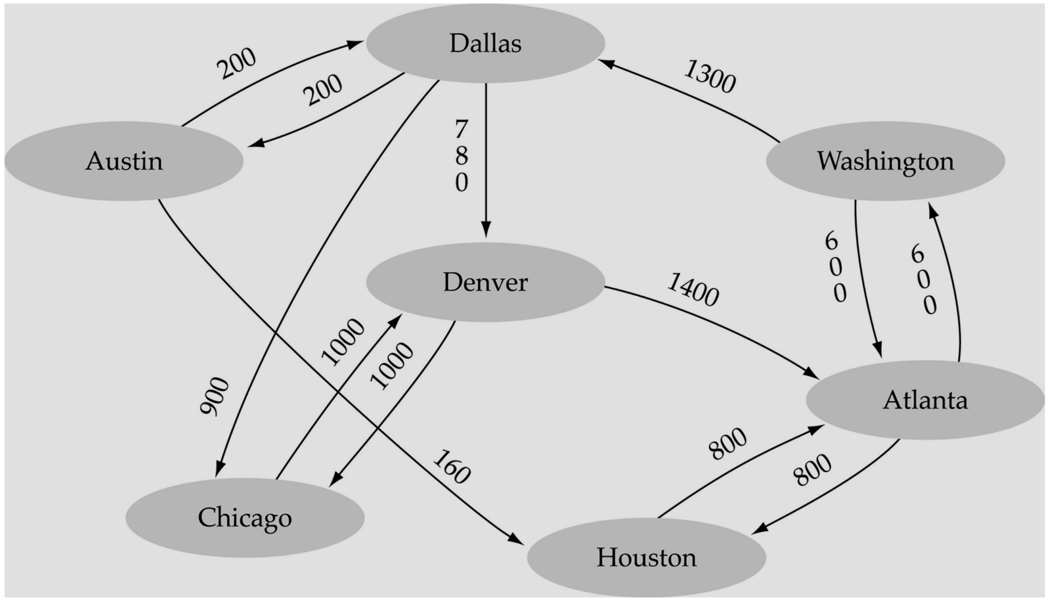
(Array positions marked '•' are undefined)

Graph implementation (cont.)

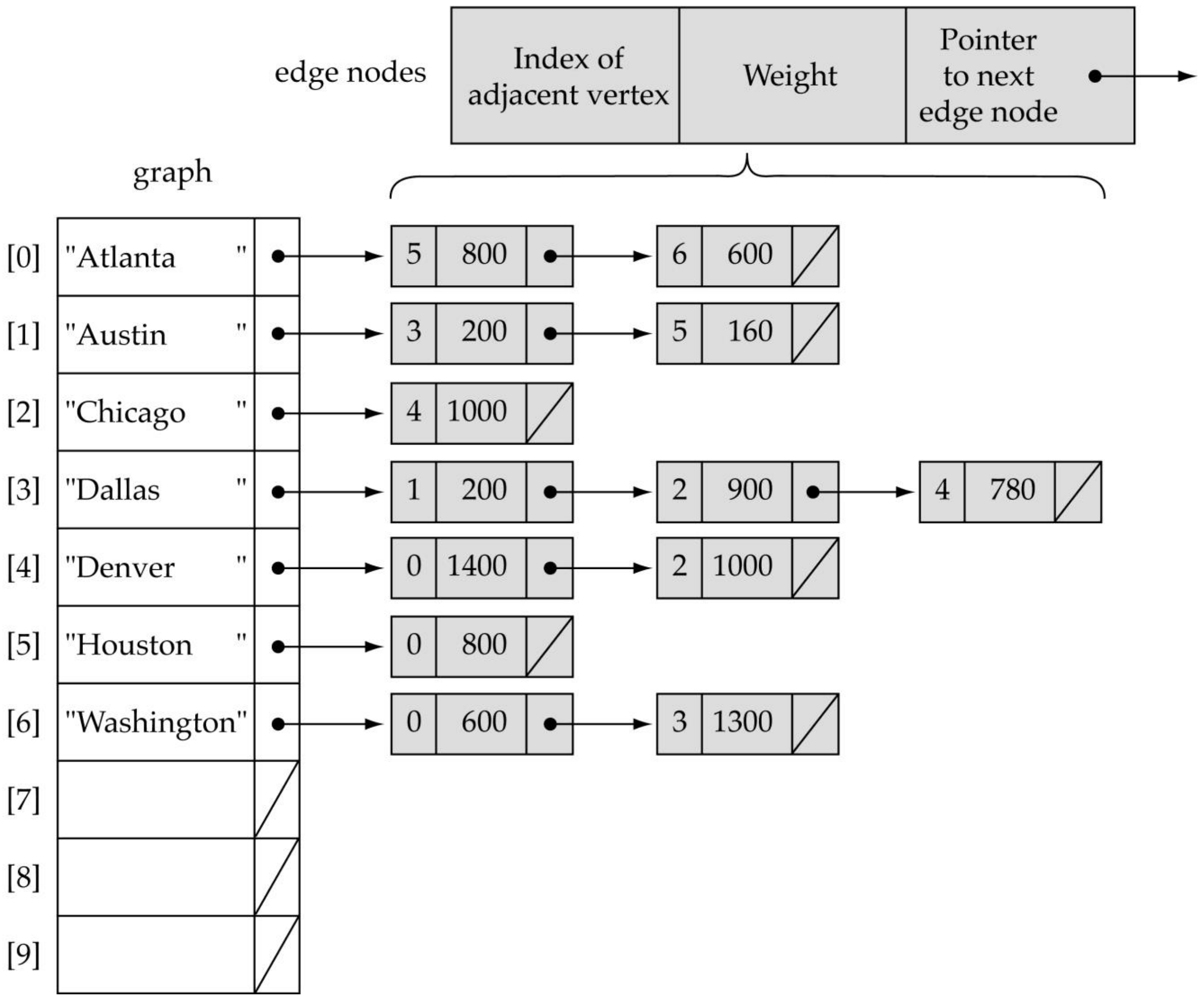
Linked-list implementation

A 1D array is used to represent the vertices

A list is used for each vertex *v* which contains the vertices which are adjacent from *v* (adjacency list)



(a)


edge nodes
Index of adjacent vertex
Weight
Pointer to next edge node
graph
[0] "Atlanta "
[1] "Austin "
[2] "Chicago "
[3] "Dallas "
[4] "Denver "
[5] "Houston "
[6] "Washington"
[7]
[8]
[9]
5 800
6 600
3 200
5 160
4 1000
1 200
2 900
4 780
0 1400
2 1000
0 800
0 600
3 1300

Adjacency matrix vs. adjacency list representation

Adjacency matrix

Good for dense graphs --$|E| \sim O(|V|^2)$

Memory requirements: $O(|V| + |E|) = O(|V|^2)$

Connectivity between two vertices can be tested quickly

Adjacency list

Good for sparse graphs -- $|E| \sim O(|V|)$

Memory requirements: $O(|V| + |E|)=O(|V|)$

Vertices adjacent to another vertex can be found quickly

Graph searching

Problem: find a path between two nodes of the graph

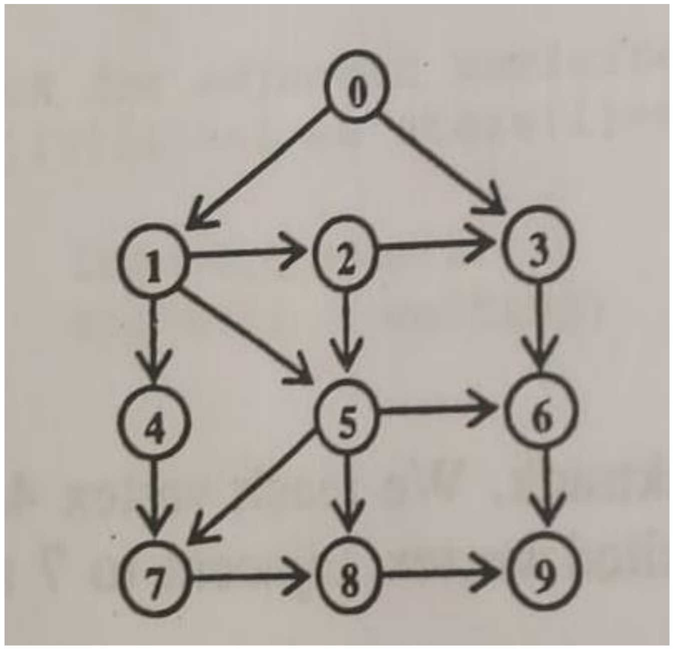
Methods: Depth-First-Search (DFS) or Breadth-First-Search (BFS)

Depth-First-Search (DFS)

Algorithm steps for DFS : Initially stack is empty

1. Push the starting node in the stack.
2. Loop until the stack is empty.
3. Pop the node from the stack inside loop.
4. If popped vertex is in initial state, visit it and change its state to visited.
5. Push all unvisited vertices adjacent to the popped vertex.
6. Repeat steps 3 to 5 until the stack is empty.

Note: There is no restriction on the order in which the successors of a vertex are visited.



Start vertex is 0, so push 0

Pop 0:	visit 0	push 3, 1	stack 3, 1
Pop 1:	visit 1	push 5, 4, 2	stack 3, 5, 4, 2
Pop 2:	visit 2	push 5, 3	stack 3, 5, 4, 5, 3
Pop 3:	visit 3	push 6	stack 3, 5, 4, 5, 6
Pop 6:	visit 6	push 9	stack 3, 5, 4, 5, 9
Pop 9:	visit 9		stack 3, 5, 4, 5
Pop 5:	visit 5	push 8, 7	stack 3, 5, 4, 8, 7
Pop 7:	visit 7	push 8	stack 3, 5, 4, 8, 8
Pop 8:	visit 8		stack 3, 5, 4, 8
Pop 8:			stack 3, 5, 4
Pop 4:	visit 4		stack 3, 5
Pop 5:			
Pop 3:			

DFS traversal : 0 1 2 3 6 9 5 7 8 4

Breadth-First-Searching (BFS)

Algorithm steps for BFS : Initially queue is empty, and all vertices are at initial state.

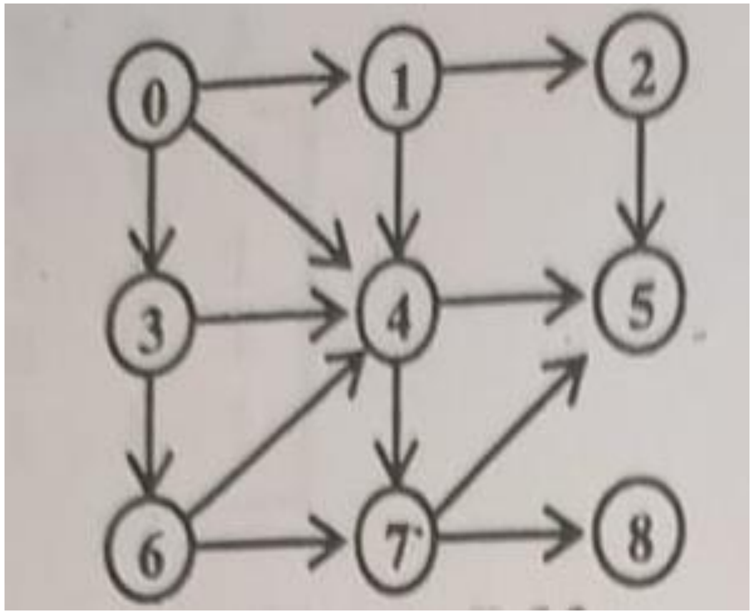
1. Insert the starting node into the queue, change its state to waiting

2. Loop until the Queue is empty.

3. Remove a node from the queue inside loop, visit it and change its state to visited

4. Look for the adjacent vertices of the deleted node, and insert only those vertices in the queue, which are at initial state. Change the state all these vertices to waiting

6. Repeat steps 3 to 5 until Queue is empty



Start vertex is 0, so insert 0 in queue

Remove 0:	visit 0	insert 1, 3, 4	queue 1, 3, 4
Remove 1:	visit 1	insert 2	queue 3, 4, 2
Remove 3:	visit 3	insert 6	queue 4, 2, 6
Remove 4:	visit 4	insert 5, 7	queue 2, 6, 5, 7
Remove 2:	visit 2	insert	queue 6, 5, 7
Remove 6:	visit 6	insert	queue 5, 7
Remove 5:	visit 5	insert	queue 7
Remove 7:	visit 7	insert 8	queue 8
Remove 8:	visit 8	insert	queue EMPTY

Here 3 states are maintained, initial, waiting and visited. The vertices that are in waiting or visited state are not inserted in queue.

Thanks