Data Structures & Algorithms Design- SS ZG519
Lecture - 14

Dr. Padma Murali

BITS Pilani
Pilani Campus

Lecture 14 Topics

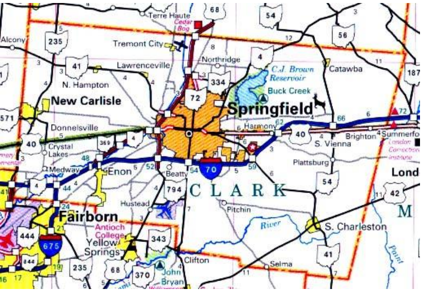
- Graph Algorithms- Introduction
- Graph Traversals- Depth First Search & Breadth First Search

BITS Pilani
Pilani Campus

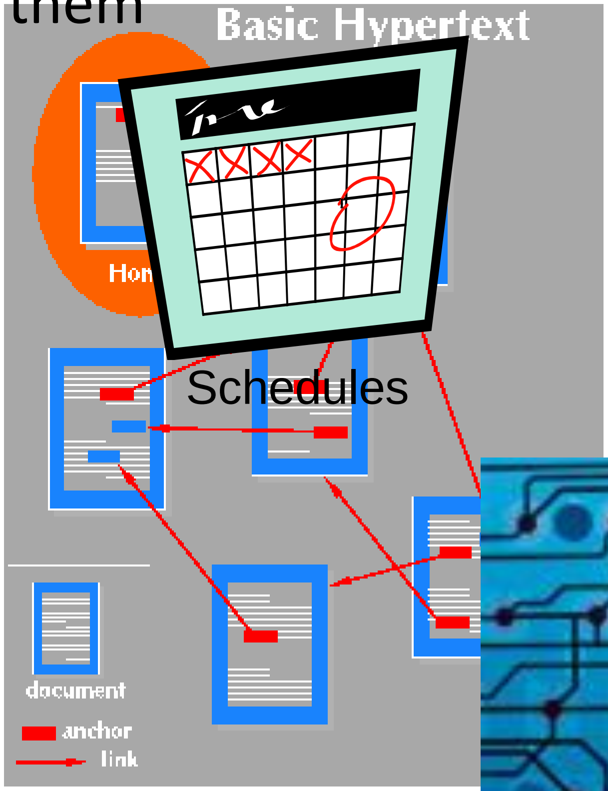
- **Graphs**

Graphs

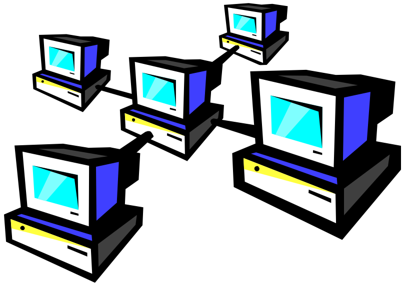
- Applications that involve not only a set of items, but also the connections between them



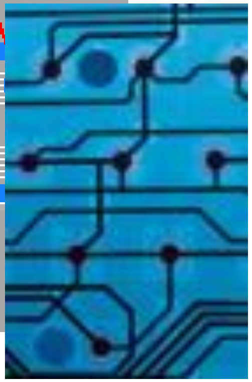
Maps



Schedules



Computer networks



Hypertext

Circuits

Graphs - An Introduction to Graphs

Graph theory is considered to have begun in the year 1736.

Graphs are used for modeling a wide variety of real life applications.

Some of the applications of graph theory are in

1. Scheduling problems- For Example: Project Scheduling

2. Computer networks, communication networks

More applications of graphs

3. Circuits

4. Hypertext

5. Maps

6. Games

7. Web's diameter- which is the maximum number of links one needs to follow to reach one web page from another by the most direct route between them.

What is a Graph ?

Example:

Four students- A, B, C, D have completed their course work in BITS, Pilani.

They are applying for their final project internship.

There are 4 different organisations: C1, C2, C3, C4 which offer projects.

The following are the preferences of the students.

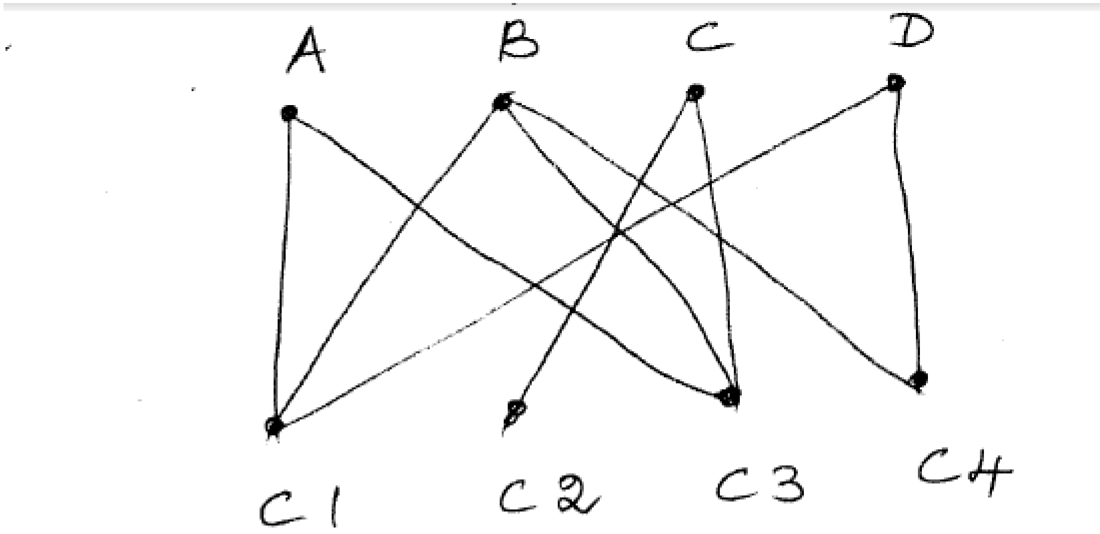
A prefers C1 and C3.

B prefers C1, C3, and C4.

C prefers C2 and C3.

D prefers C1 and C4.

The preferences of the students can be diagrammatically represented in the following manner.



The above diagram is called a Graph.

Graph:

A graph G is a finite non empty set V(G) of objects called **vertices (also called as nodes)** and a set E(G) of two element subsets of V(G) called **edges.**

V(G) is called the **vertex set** of graph G and

E(G) is called the **edge set of graph G.**

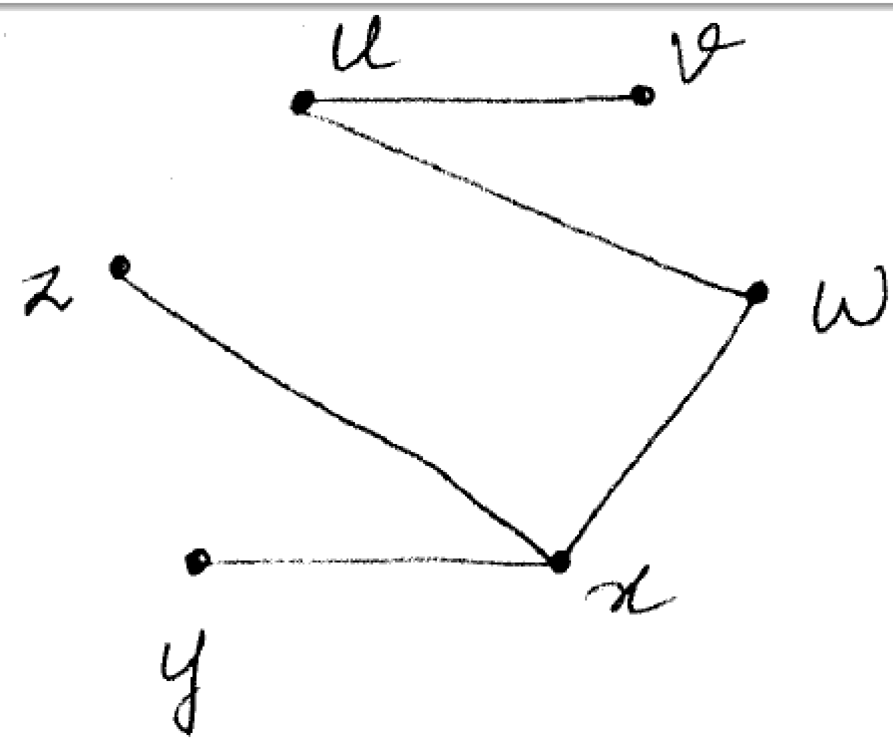
Let G be a graph and {u, v}, an edge of G. Since,{u, v} is a 2-element set, we may write {v, u} instead of {u, v}. Conveniently, we represent this edge by uv or vu.

<u>**Adjacent Vertices:**</u> If e = vu is an edge of a graph G, then we say that u and v are adjacent in G.

<u>**Example:**</u> A graph G is defined by the sets

V(G) = {u,v,w,x,y,z}

E(G) = {uv,uw,wx,xy,xz}

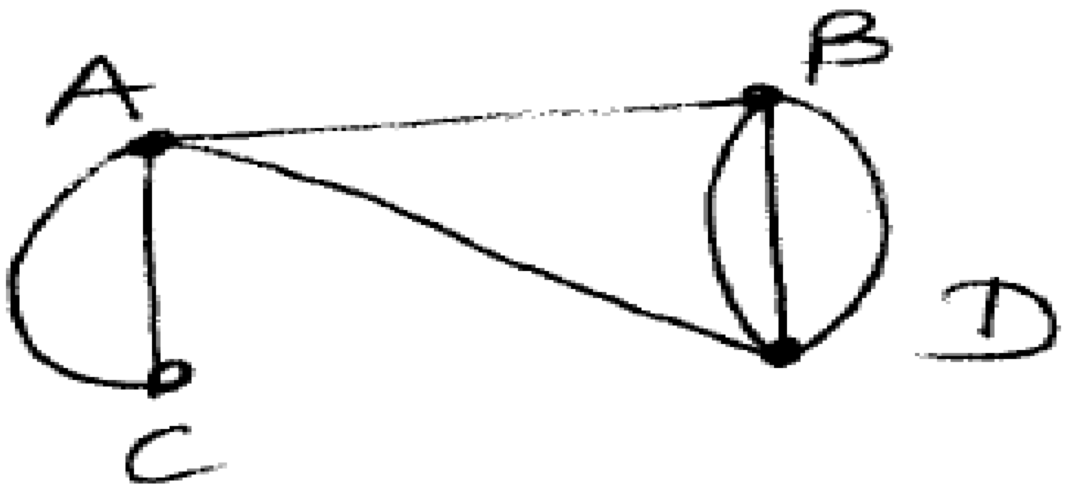


Every graph has a diagram associated with it. The diagrams are useful for understanding problems involving such a graph. The vertices are represented by means of points and by joining two points by means of a line segment is an edge.

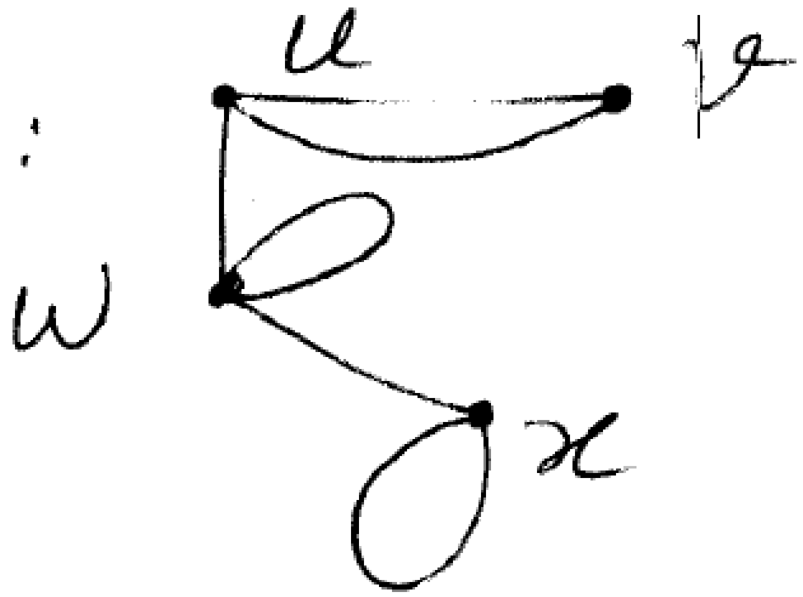
Parallel Edges: Two or more edges that join the same pair of vertices are called parallel edges.

Example: In a road network, more than one road may connect a pair of cities.

Multigraph: If in a graph, there are parallel edges, such a graph is called a multigraph.



Loop: An edge that joins a vertex to itself is called a loop.



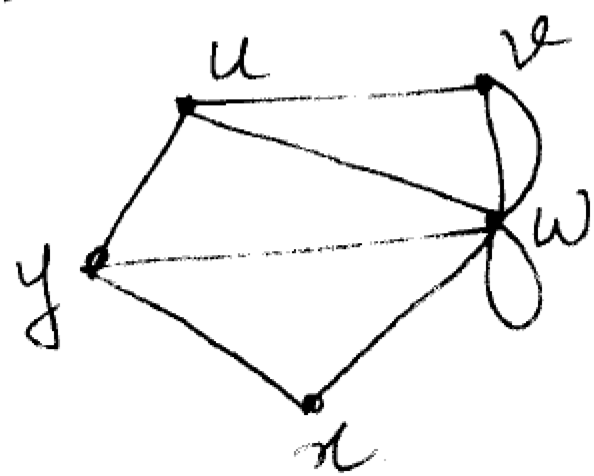
<u>Psuedograph:</u> A graph that contains both parallel edges and loops is called a psuedograph.

<u>Order:</u> The number of vertices in a graph G is called its order.

<u>Size:</u> The number of edges in a graph G is called its size.

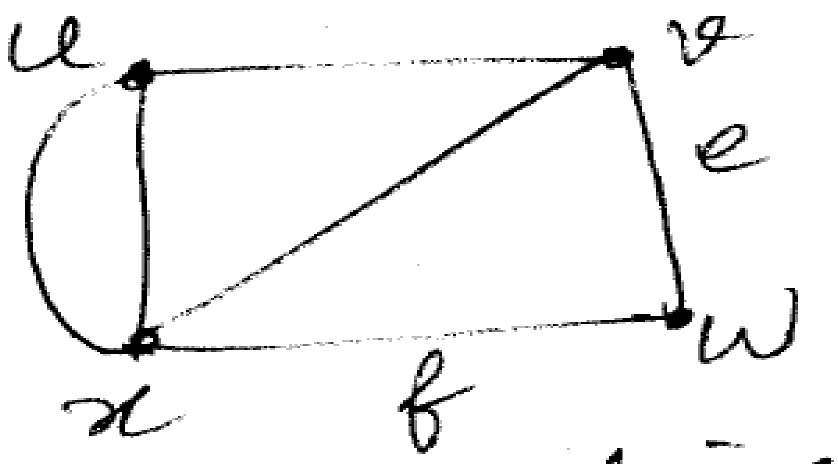
<u>Degree of a vertex:</u> The number of edges incident on a vertex is called the degree of a vertex.

Example:



$\deg(u) = 3$
$\deg(v) = 3$
$\deg(w) = 7$
$\deg(x) = 2$
$\deg(y) = 3.$

Adjacent Edges: If e and f are distinct edges that are incident with a common vertex, then e and f are adjacent edges.



Order of graph = 4
Size of graph = 6

e and f are adjacent edges and u and v are adjacent vertices.

Isolated vertex : A vertex of degree 0 is called an isolated vertex.

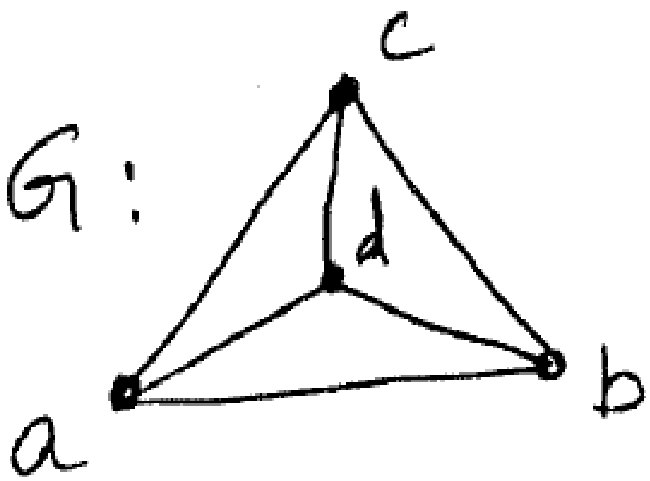
End vertex: A vertex of degree 1 is called an end vertex.

Even vertex: A vertex is called even if the degree of the vertex is even.

Odd vertex: A vertex is called odd if the degree of the vertex is odd.

Regular Graph: A graph G is r-regular or regular of degree r, If every vertex of G has degree r.

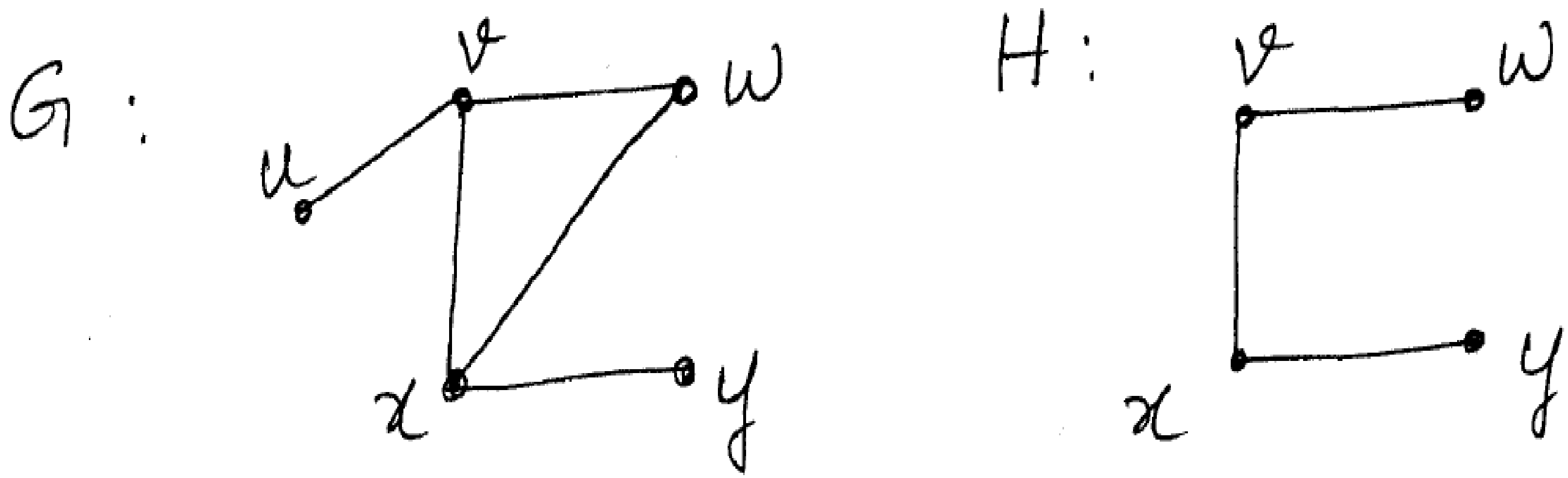
Example:



G is a regular graph of degree 3.

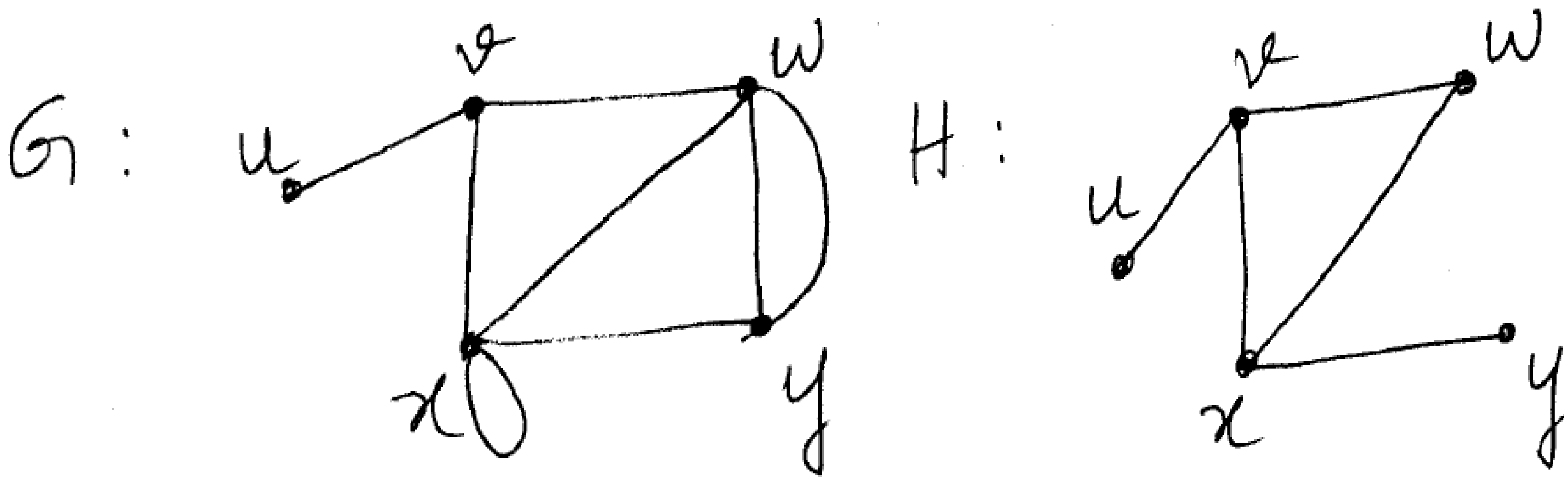
Subgraph: A graph H is a subgraph of a graph G if

$$V(H) \subseteq V(G) \text{ and } E(H) \subseteq E(G).$$



H is a subgraph of G.

Spanning subgraph: A subgraph H of a graph G is a spanning subgraph of G if V(H) = V(G).



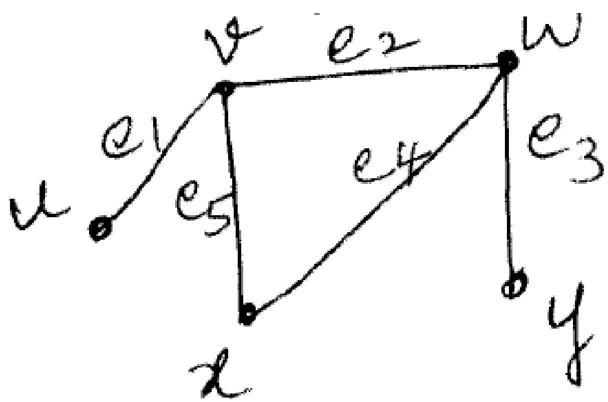
H is a spanning subgraph of G.

Walk: A walk in a graph G is an alternating sequence of vertices and edges.

$$W : v_0, e_1, v_1, e_2, v_2 \ldots\ldots e_n, v_n$$

Example :

G :



W: u, e1, v, e5, x, e4, w is a walk

Length of a walk: A walk is of length n if it has n edges.
In the previous example, walk

$W: u, e_1, v, e_5, x, e_4, w$

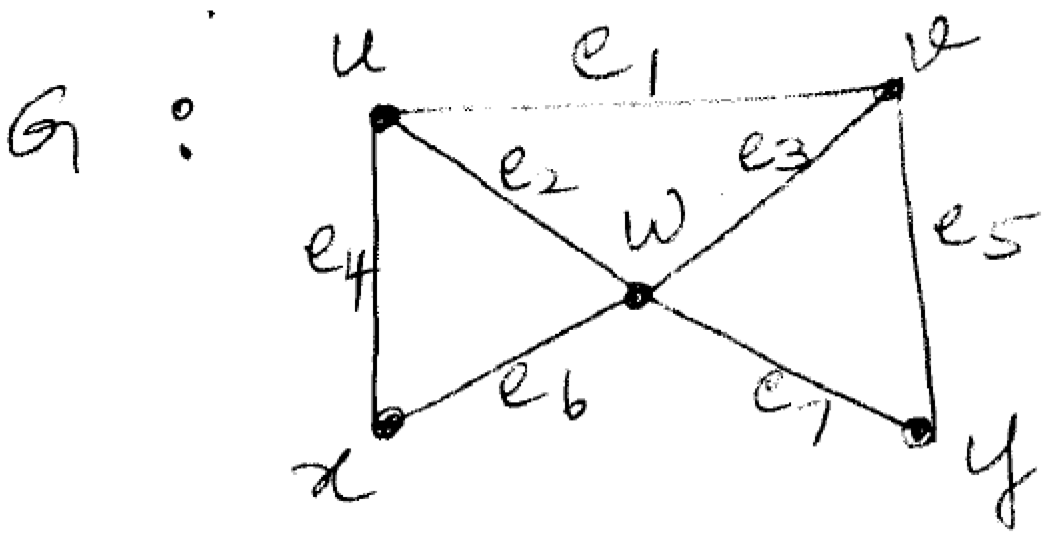
is a walk of length 3.

Trivial walk: A walk of length zero is called a trivial walk.

Trail: A trail is a walk in which no edge is repeated.

<u>Path:</u> A path is a walk in which no vertex is repeated.

<u>Example:</u>



In G, the walk x, w, v, u, w, y is a trail that is not a path.

u x w v y is a path.

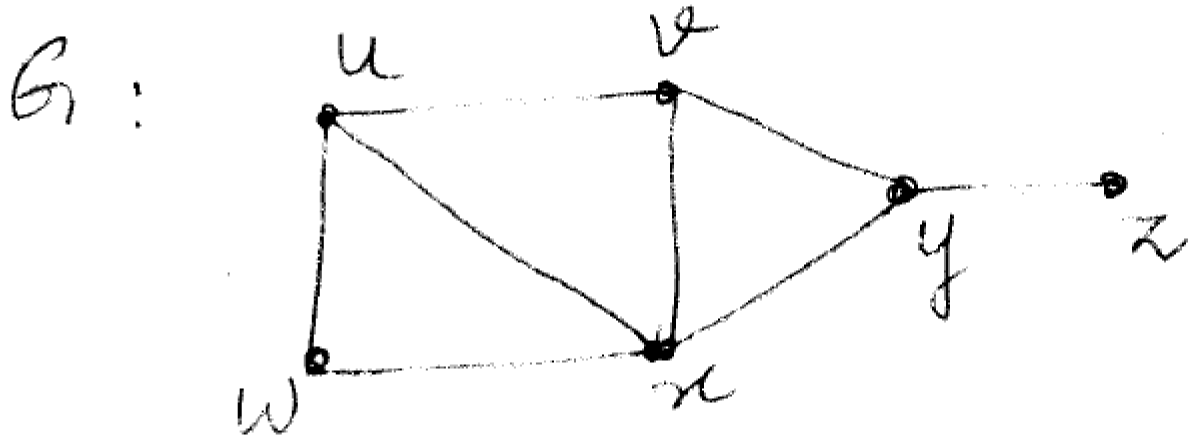
Cycle or Circuits: A cycle is a walk $V_0, V_1, V_2 \ldots\ldots V_n$

in which n >= 3, $V_0 = V_n$ and the n vertices

$V_1, V_2 \ldots\ldots V_n$ are all distinct.

A cycle of length n is referred to as an n-cycle.

Example:

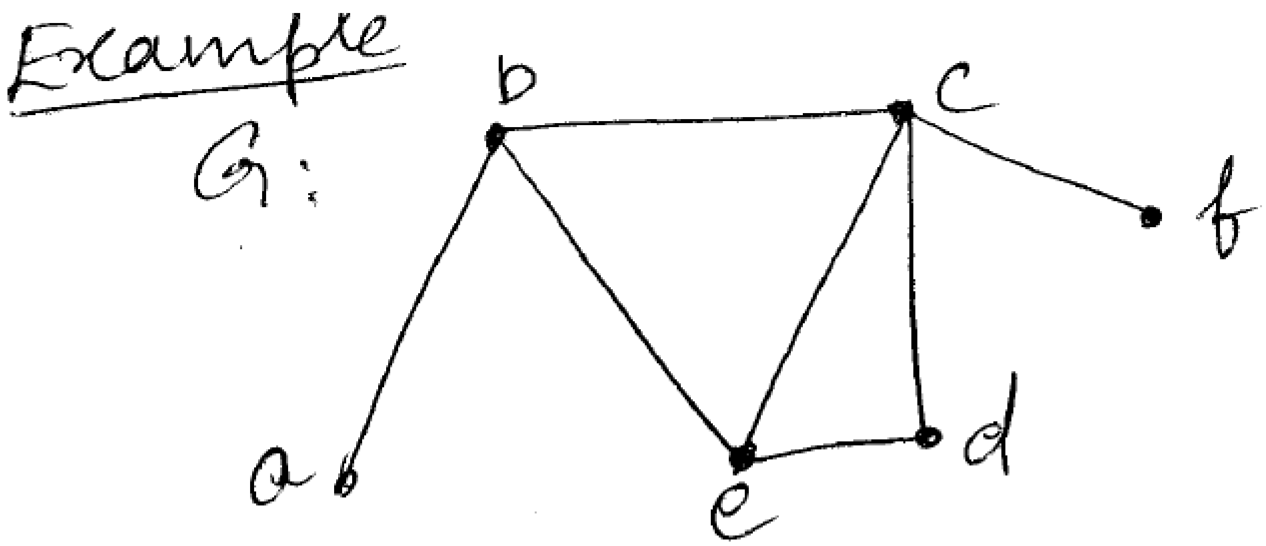


In G, u-v-x-w-u is a cycle of length 4.

Connected: Let u and v be vertices in a graph G. We say that u is connected to v if G contains a u-v path.

Connected graph: The graph G is connected if u is connected to v for every pair u,v of vertices of G.

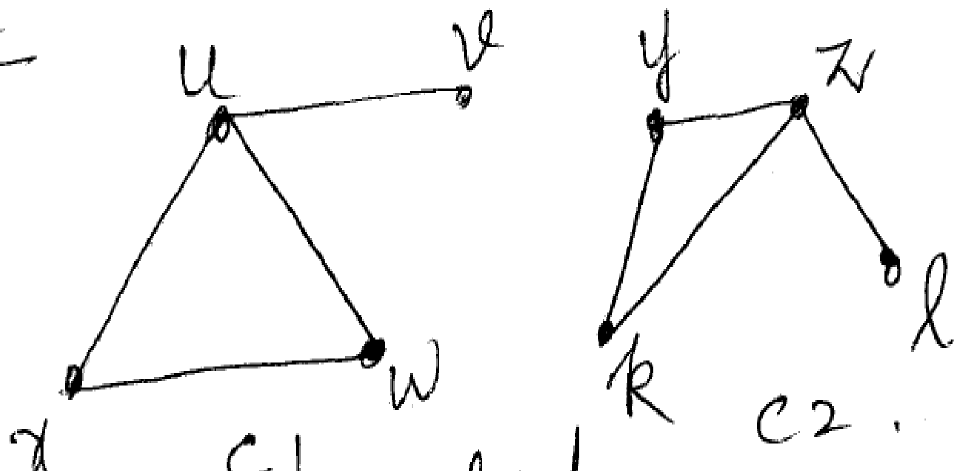
A graph that is not connected is called a disconnected graph.



G is a connected graph.

Example

G :



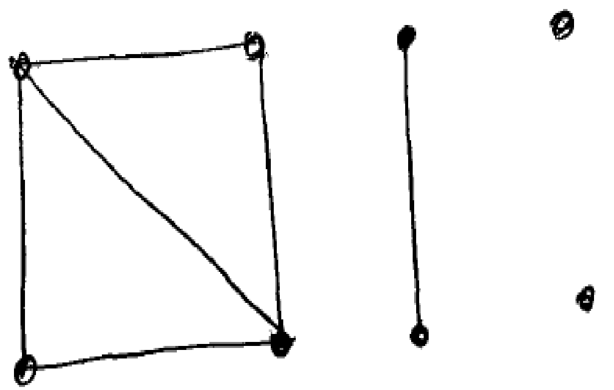
G is disconnected.

C1 and C2 are called components of G.

<u>Components:</u> The maximal connected subgraphs of a Disconnected graph G are called its components.

Example:

G:

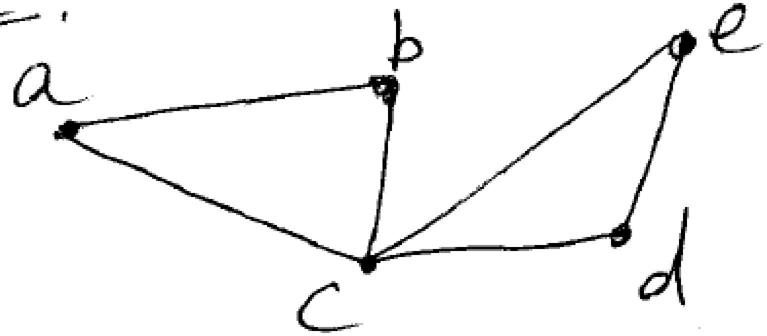


G is a disconnected graph with 4 components

Cut vertex: A vertex v in a connected graph G is called a cut vertex if G – v is disconnected. In graph G, G-{c} is disconnected. Hence, c is a cut vertex.

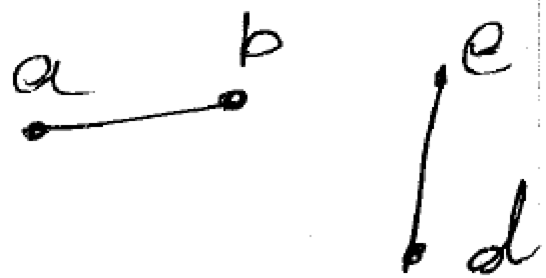
Example:

G:



a.

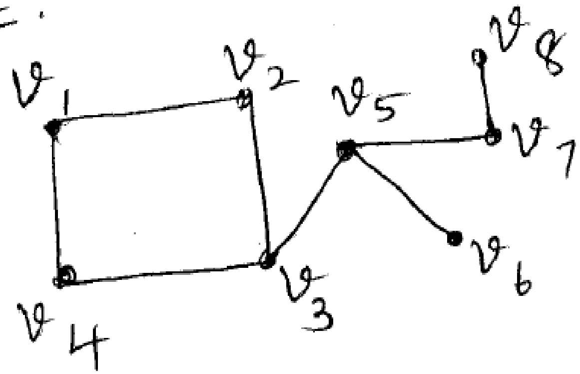
G-{c}



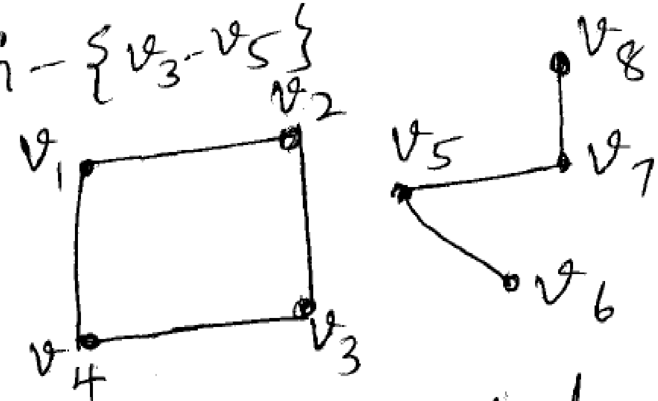
Bridge: An edge ‘e’ in a connected graph G is called a bridge if G – e is disconnected.

Example:

G :



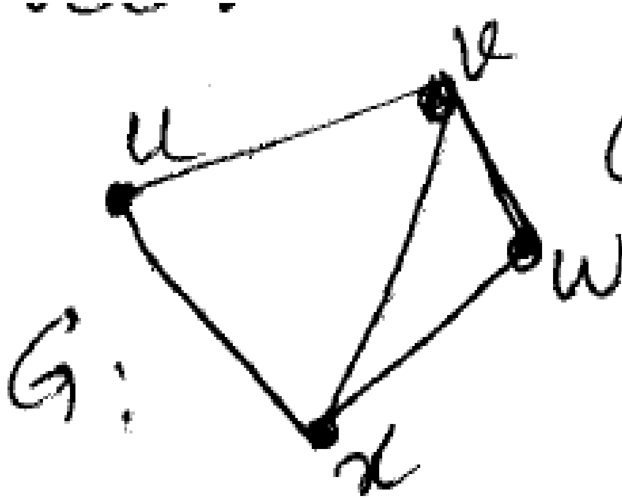
$G - \{v_3 - v_5\}$



In graph G, $G - \{v_3 - v_5\}$ is disconnected

$\therefore$ $\{v_3 - v_5\}$ edge is a bridge.

<u>Non-separable graph:</u> A nontrivial connected graph without a cut vertex is called a non-separable graph.

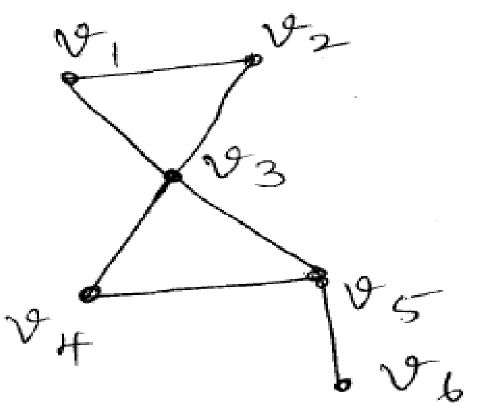


G is a nonseparable graph.

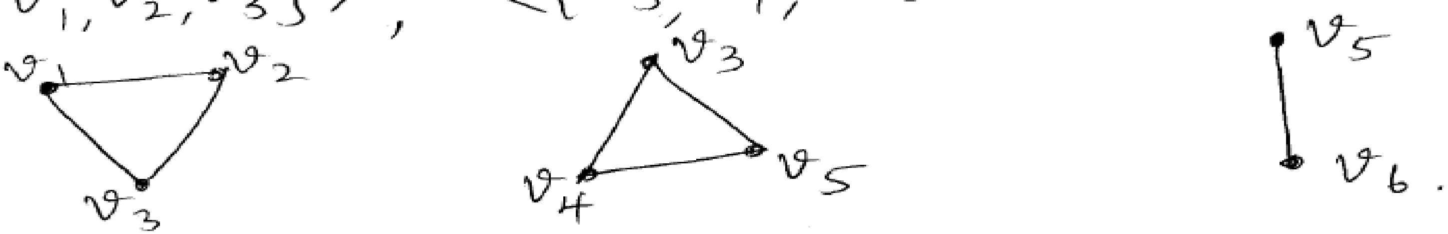
Blocks: Let G be a non trivial connected graph. A block of G is a subgraph of G that is itself a maximal non separable graph.

Example:

G :



Graph G has 3 blocks namely $\langle\{v_1, v_2, v_3\}\rangle$, $\langle\{v_3, v_4, v_5\}\rangle$ and $\langle\{v_5, v_6\}\rangle$



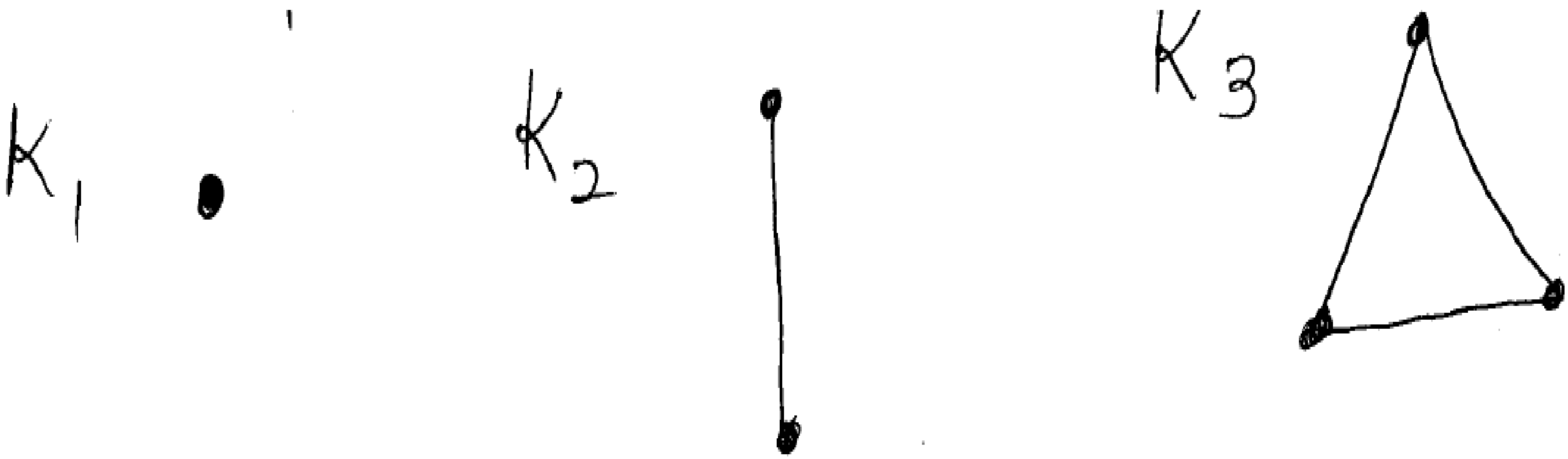
Note:

1. The blocks of a graph produce a partition of the edge set of the graph.

2. Every two blocks have at most one vertex in common.

3. If two blocks share a vertex, then the vertex is a cut vertex.

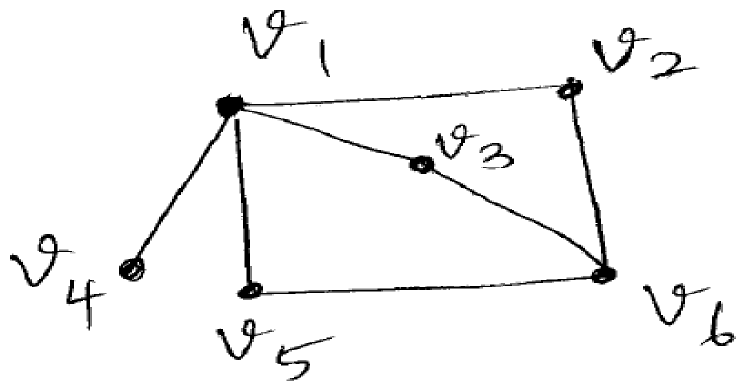
Complete graphs: A graph in which every distinct pair of vertices are adjacent is called a complete graph.
K_n denotes the complete graph on n vertices.



Bipartite graph: A graph G is called bipartite if the vertex set V(G) of G can be partitioned into two non empty subsets V_1 and V_2 such that every edge of G joins a vertex of V_1 and a vertex of V_2

Example:

G :



G is bipartite $V_1 = \{v_1, v_6\}$

$V_2 = \{v_2, v_3, v_4, v_5\}$

Representation of Graphs

Adjacency Matrix: Let G be a (p,q) graph with p vertices and q edges. $V(G) = \{ v_1, v_2, \ldots v_p \}$

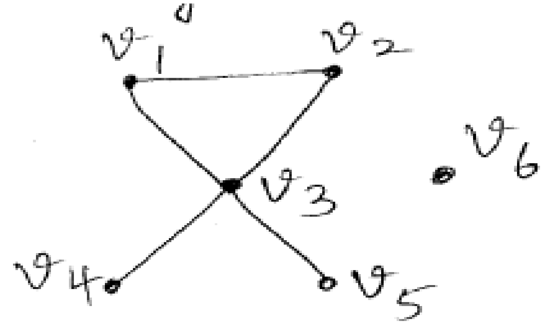
The adjacency matrix $A = [a_{ij}]$ of G is the p X p matrix defined by $a_{ij} = \{1,\ \text{if } v_i v_j \in E(G)$

$\{ 0,$ otherwise

Thus A is a symmetric matrix in which every entry on the main diagonal is 0.

Example:

G :



$$V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6\}$$

$$E(G) = \{v_1v_2, v_1v_3, v_2v_3, v_3v_4, v_3v_5\}$$

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{cccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{array} \\ \left[\begin{array}{cccccc} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}_{6 \times 6}$$

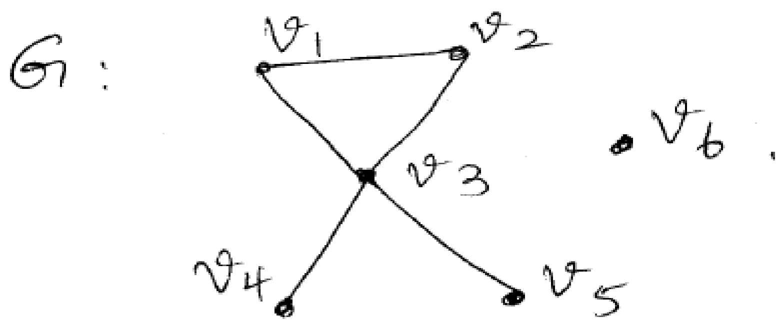
Since, A is a $p \times p$ matrix, p^2 memory locations need to be allocated for its entries.

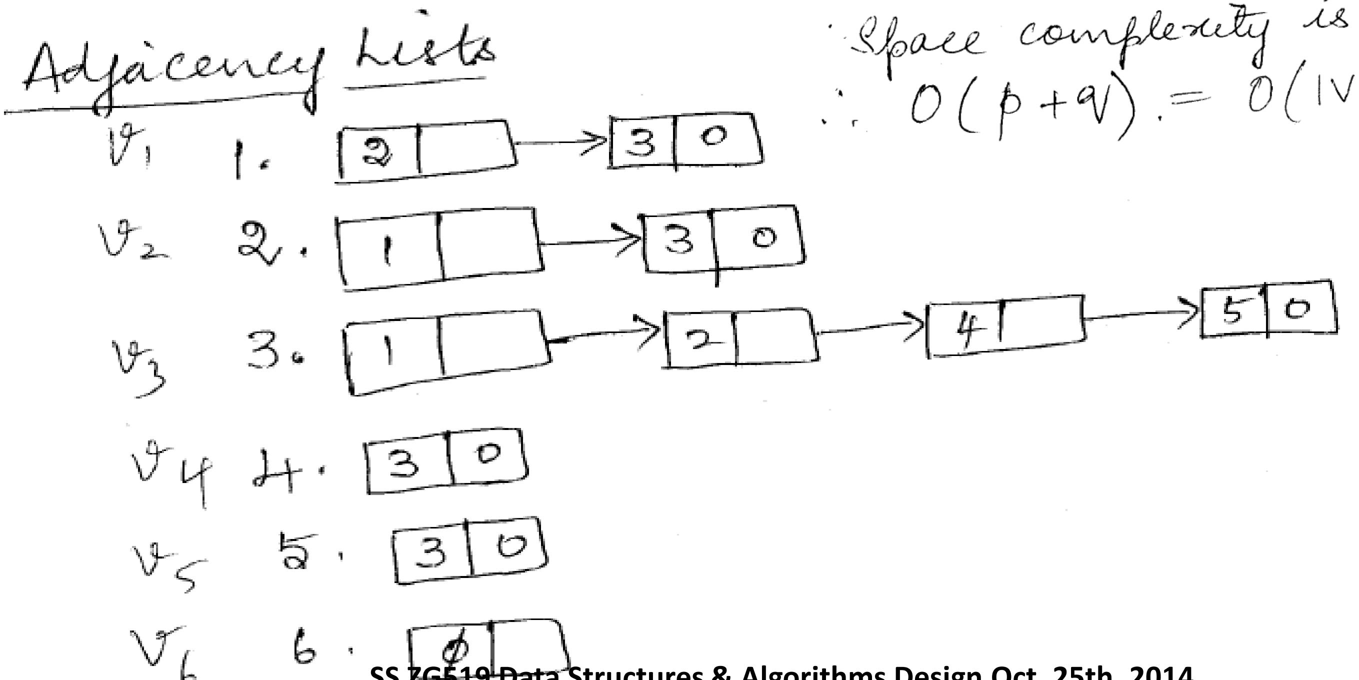
Note:

If G is a graph with relatively few edges, then many locations of its adjacency matrix contain 0. Thus, unusually Large amount of memory space is required for relatively few edges.

Adjacency Lists: Let G be a graph with vertex set $V(G) = \{ v_1, v_2, \ldots.. v_p \}$. The adjacency list representation of G associates with each vertex a list of its adjacent vertices.

Example:





∴ Space complexity is $p+2q$ locations

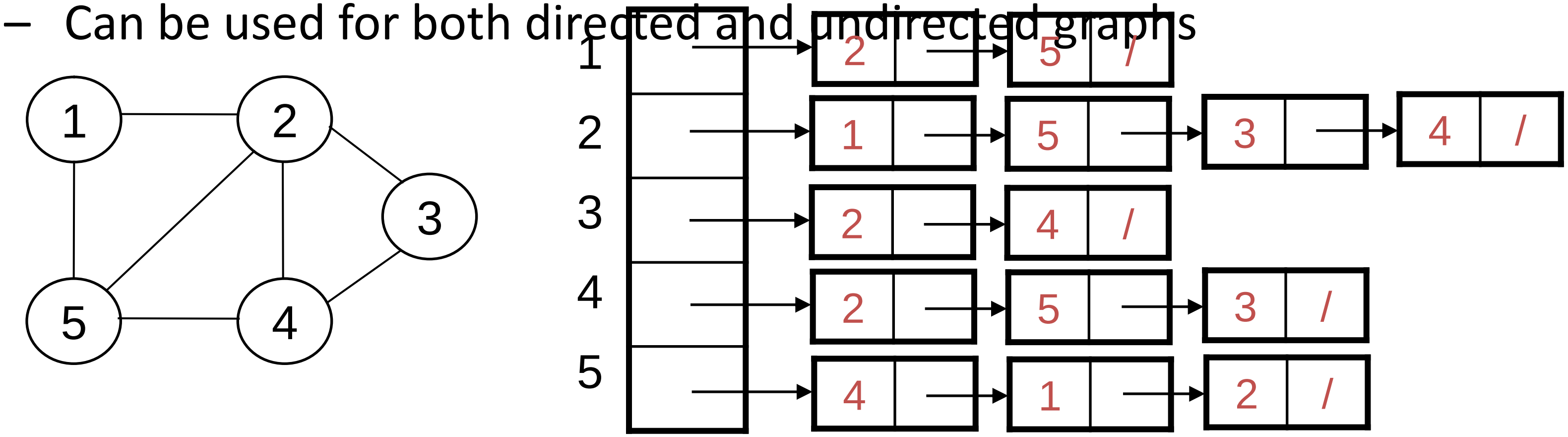
∴ $O(p+q) = O(|V|+|E|)$

Graph Representation

innovate achieve lead

Adjacency list representation of G = (V, E)

- An array of |V| lists, one for each vertex in V
- Each list Adj[u] contains all the vertices v such that there is an edge between u and v
  - Adj[u] contains the vertices adjacent to u (in arbitrary order)
- Can be used for both directed and undirected graphs



Undirected graph

Properties of Adjacency-List Representation

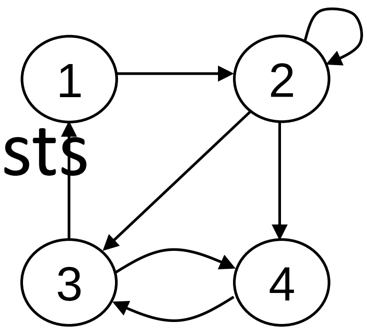
Sum of the lengths of all the adjacency lists

$|E|$

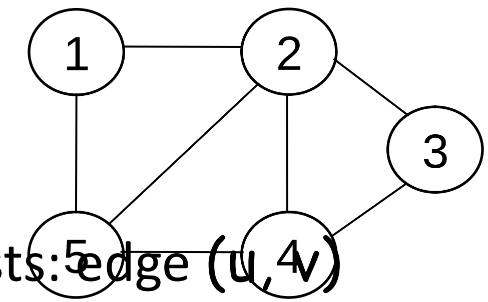
- Directed graph:
  - Edge (u, v) appears only once in u's list

$2|E|$

- Undirected graph:
  - u and v appear in each other's adjacency lists: edge (u, v) appears twice

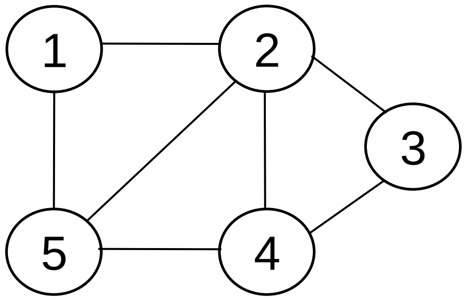


Directed graph



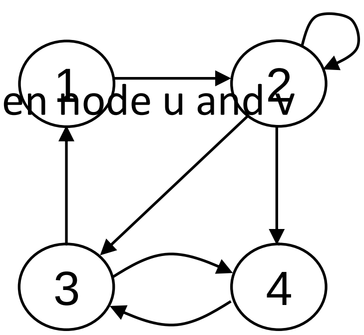
Undirected graph

Properties of Adjacency-List Representation



Undirected graph

- Memory required
  - $\Theta(V + E)$
- Preferred when
  - the graph is sparse: $|E| << |V|^2$
- Disadvantage
  - no quick way to determine whether there is an edge between node u and v
- Time to list all vertices adjacent to u:
  - $\Theta(\text{degree}(u))$
- Time to determine if $(u, v) \in E$:
  - $\Theta(\text{degree}(u))$

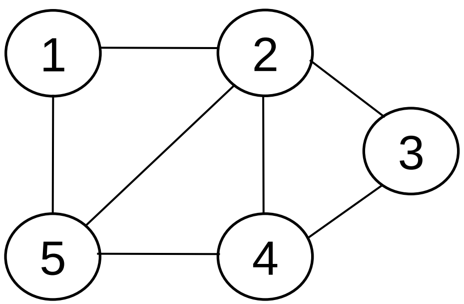


Directed graph

Graph Representation

Adjacency matrix representation of G = (V, E)

- Assume vertices are numbered 1, 2, … |V|
- The representation consists of a matrix $A_{|V| x |V|}$:
- $a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$



Undirected graph

	1	2	3	4	5
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

For undirected graphs matrix A is symmetric:

$a_{ij} = a_{ji}$

$A = A^T$

Properties of Adjacency Matrix Representation

Memory required

- $\Theta(V^2)$, independent on the number of edges in G

Preferred when

- The graph is dense $|E|$ is close to $|V|^2$
- We need to quickly determine if there is an edge between two vertices

Time to list all vertices adjacent to u:

- $\Theta(V)$

Time to determine if $(u, v) \in E$:

- $\Theta(1)$

Weighted Graphs

- **Weighted graphs** = graphs for which each edge has an associated weight $w(u, v)$

$$w: E \rightarrow \mathrm{R}, \text{ weight function}$$

- Storing the weights of a graph
  - Adjacency list:
    - Store $w(u,v)$ along with vertex v in u's adjacency list
  - Adjacency matrix:
    - Store $w(u, v)$ at location (u, v) in the matrix

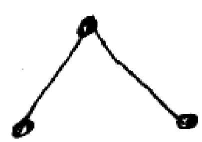
Trees: Tree is a connected graph without cycles.

Example:

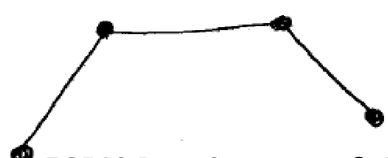
Tree of order 1, T_1

T_2

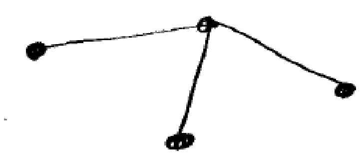
T_3



T_4



A



T_4

Remarks:

1. A tree on n vertices has n-1 edges.

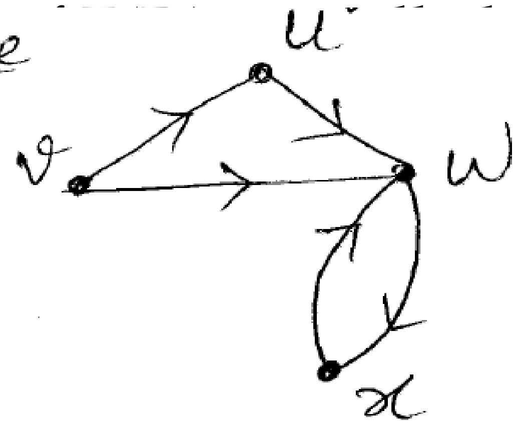
2. Every non trivial tree contains atleast two end vertices.

3. If u and v are distinct vertices of a tree T, then T contains exactly one u-v path.

Digraphs: A digraph (or directed graph) D is a finite nonempty Set V(D) of vertices and a set E(D) of ordered pairs of distinct vertices.

Example

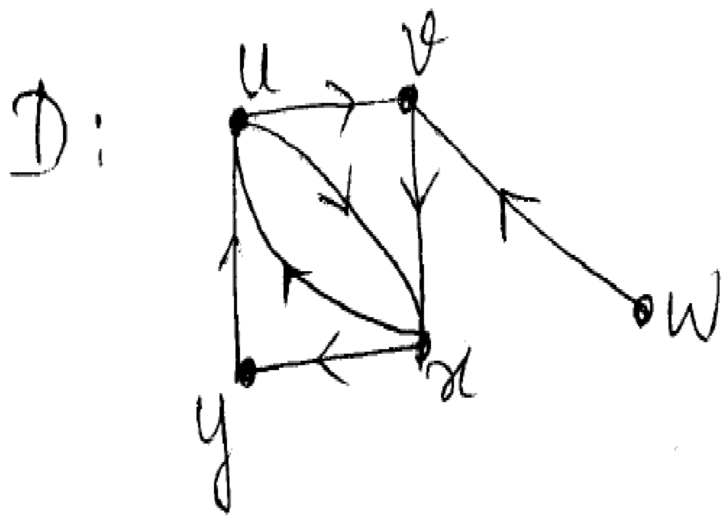
D :

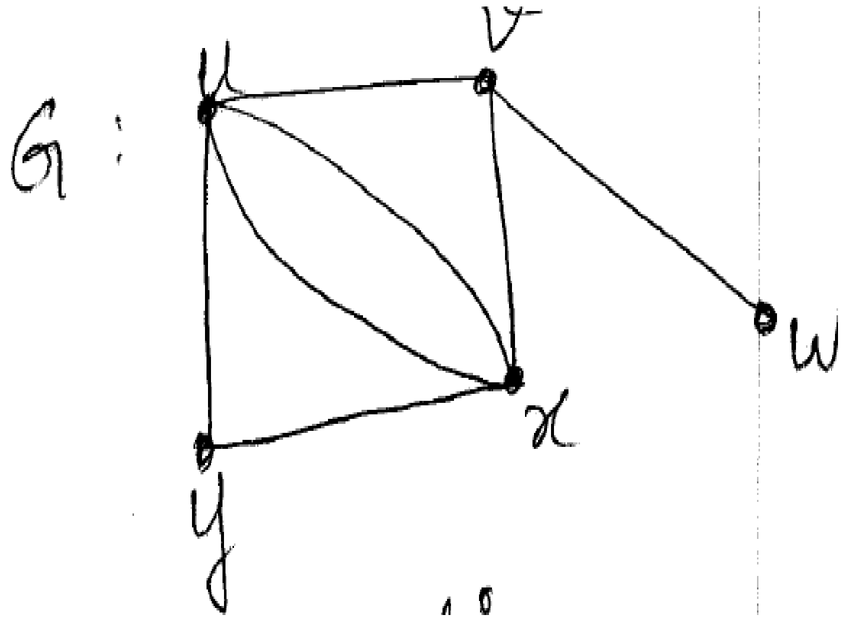


$V(D) = \{u, v, w, x\}$

$E(D) = \{(u,w), (v,u) (v,w) (x,w), (w,x)\}$

Underlying graph of a digraph D is that graph G obtained from D by replacing all arcs(u,v) or (v,u) by the edge uv.

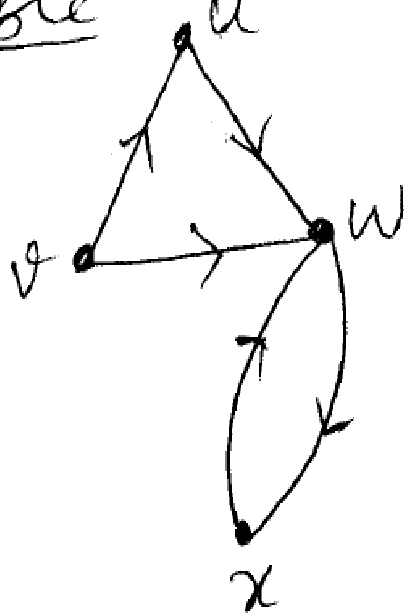




<u>Out degree</u> is the number of vertices adjacent from a vertex v.

<u>In degree</u> of a vertex v is the number of vertices adjacent to v.

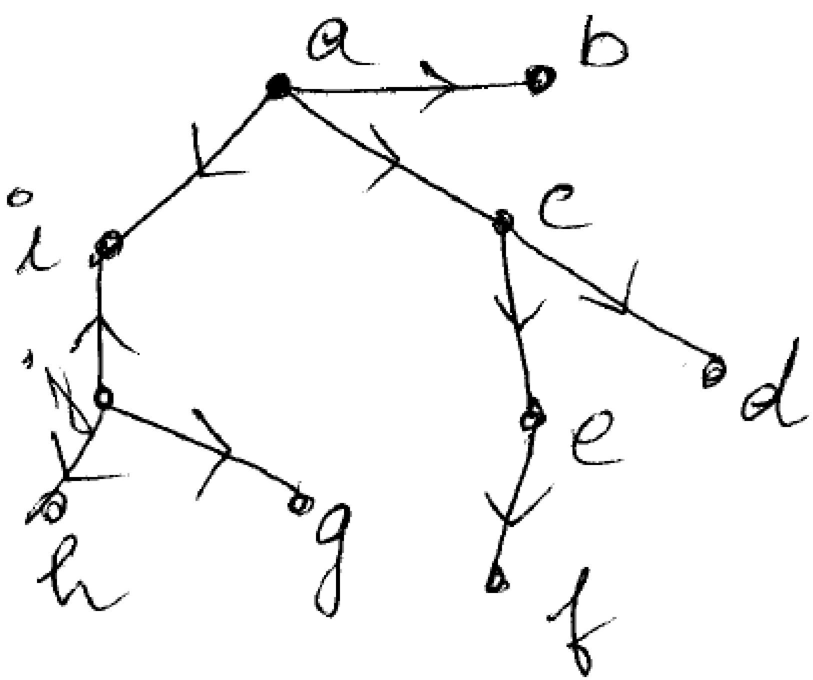
<u>Example</u>



Vertex	Outdegree	Indegree
u	1	1
v	2	0
w	1	3
x	1	1

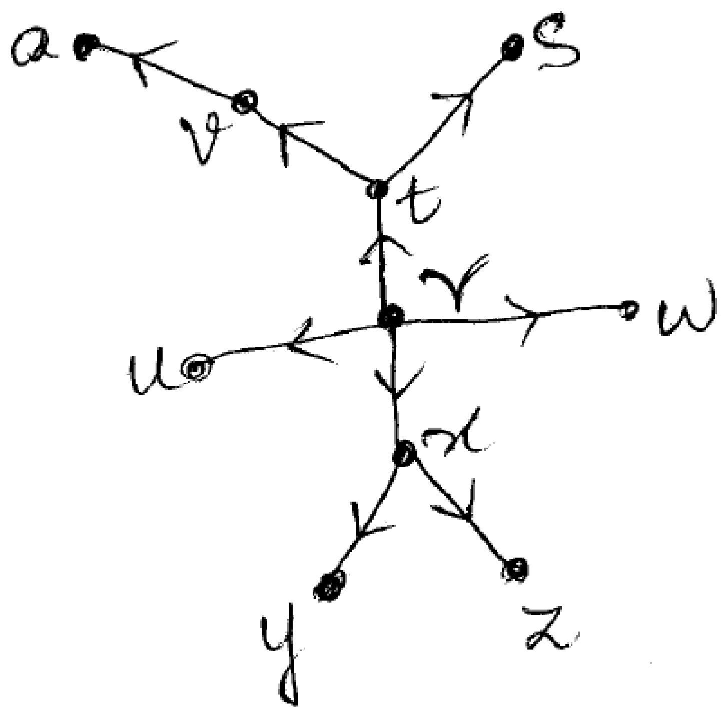
Directed Tree: A directed tree is an asymmetric digraph whose underlying graph is a tree.

Example:



Rooted Tree: A directed tree T is called a rooted tree if there exists a vertex r of T called the root such that for every vertex v of T, there is a directed r-v path in T.

Example:



Note:

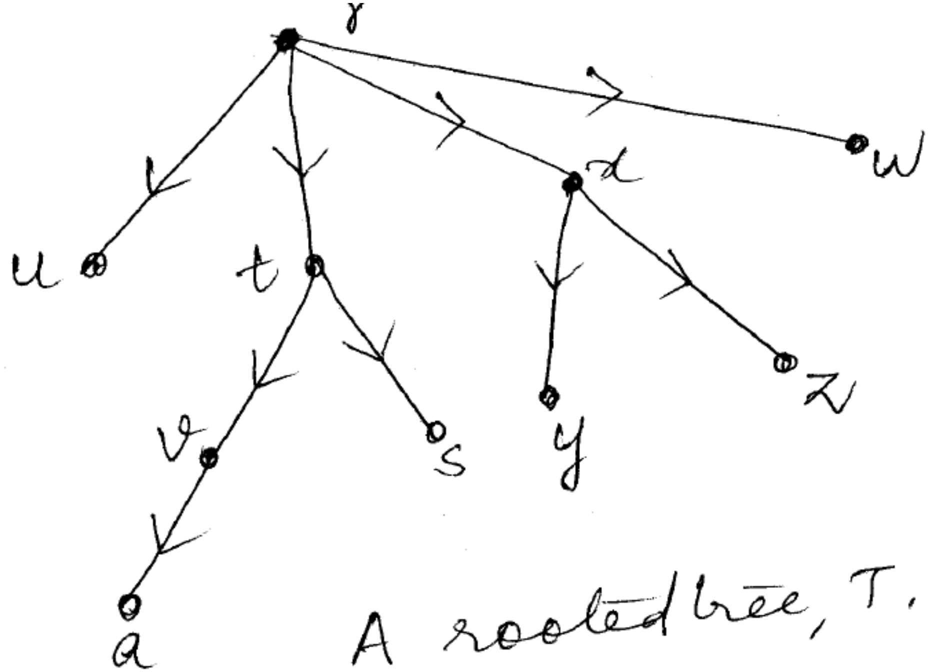
•If T is a rooted tree, then it is customary to draw T with root r at the top at level.

• The vertices adjacent from r are placed one level below at level 1.

•Any vertex adjacent from a vertex at level 1 is at level 2, and so on.

•In general, every vertex at level i > 0 is adjacent from exactly one vertex, namely one at level i-1.

Height: The largest integer h for which there is a vertex at level h in a rooted tree is called its height.



A rooted tree, T.

Height of T is 3.

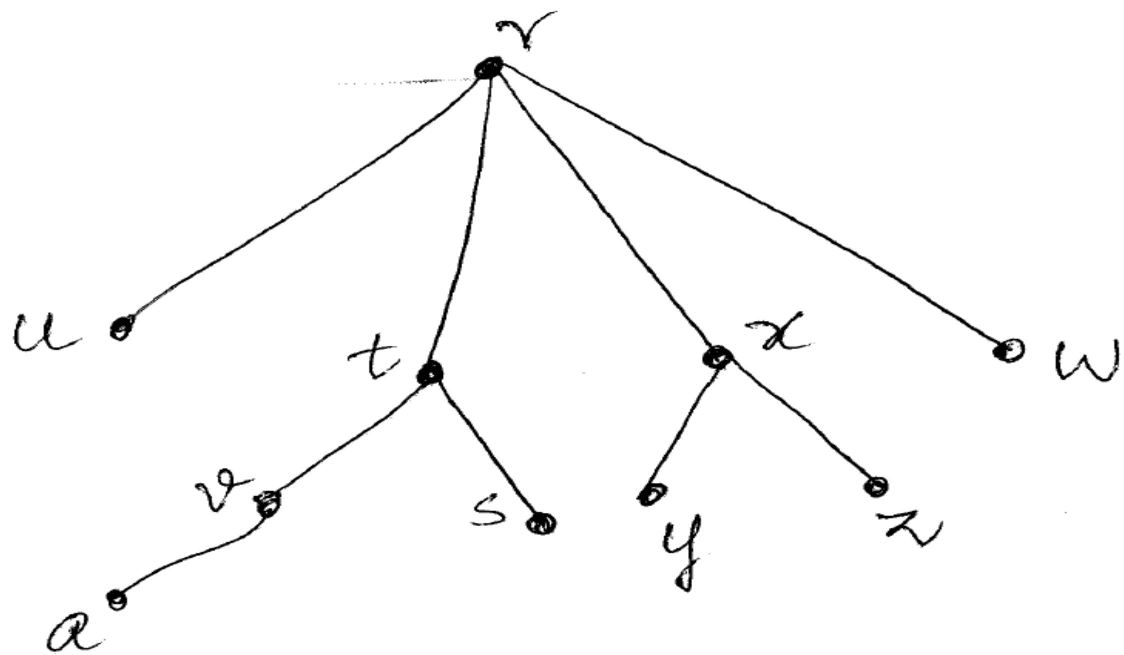
Let T be a rooted tree. If a vertex v of T is adjacent to u and u lies in the level below v, then u is called a **child** of v and v is the **parent** of u.

A vertex w is a **descendant** of v and v is an **ancestor** of w if the v-w path in T lies below v.

The vertex z is a child of x, and x is the parent of both y and z. a is a descendant of t since the t-a path t,v,a in T lies below t. But y is not a descendant of t since the t-y path t,r,x,y inT contains vertices that are not below t.

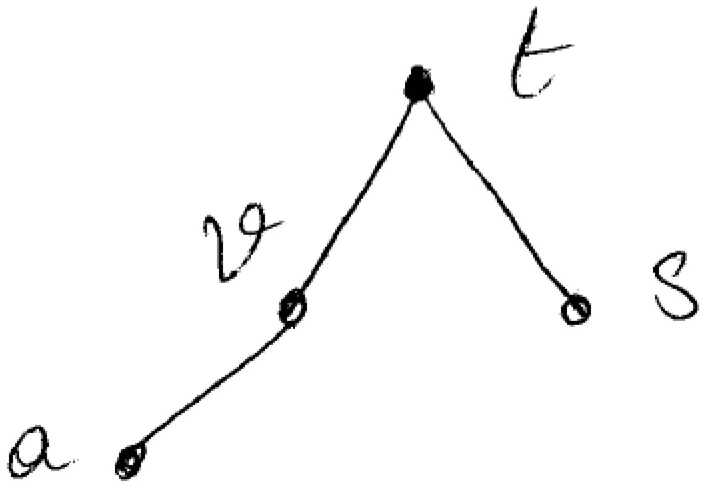
Example:

T:



Maximal Subtree: The subtree of a rooted tree T induced by a vertex v and all of its descendants is also a rooted tree with root v. This subtree is called the maximal subtree of T rooted at v.

Example



From the tree T in the previous example, this is a maximal subtree rooted at t.

Note: In a rooted tree , only the root has no parent, while every other vertex has exactly one parent.

Leaf: A vertex with no children is called a leaf; all other vertices are called **internal vertices.**

m-ary tree: A rooted tree is called m-ary if every vertex has atmost m children.

A **binary tree** is a 2-ary tree in which each child is designated as a left chid or a right child.

A rooted tree T is called a **complete m-ary tree** if every vertex of T has m children or no children.

Thus in a **complete binary tree**, every vertex has two children or no children.

A rooted tree of height h is **balanced** if every leaf is at level h or level h – 1.

BITS Pilani
Pilani Campus

Graph Searching Algorithms

Graph Searching Algorithms

- Searching a graph means systematically following the edges of the graph so as to visit the vertices of the graph.

- A graph-searching algorithm can discover much about the structure of a graph.

- Many algorithms begin by searching their input graph to obtain this structural information.

Searching in a Graph

Graph searching : systematically follow the edges of the graph so as to visit the vertices of the graph

Two basic graph searching algorithms:

- Breadth-first search
- Depth-first search
- The difference between them is in the order in which they explore the unvisited edges of the graph

Graph algorithms are typically elaborations of the basic graph-searching algorithms

Graph Searching Algorithms

Depth First Search

- Depth First Search is a powerful technique for solving many graph theory problems

- It is a systematic way of visiting all the vertices of a graph.

Description of Depth First search Traversal

1. Assume that a given graph G with $V(G) = \{ v_1, v_2, \ldots ..v_p \}$ is represented by its adjacency lists.

2. Unless indicated otherwise, we assume , in the adjacency list of a given vertex that the vertices adjacent to that vertex are listed in increasing order of their subscripts.

3. In a depth first search of G, the vertex that is currently visited is designated as the **active vertex.**

Description of Depth First search Traversal

4. **A** depth first search of G is begun by selecting a first vertex to visit namely v_1. Vertex v_1 is the first active vertex and is assigned label 1.

5. Next, select a vertex adjacent to 1 (the first vertex on the adjacency list of 1). Label it 2 and this vertex becomes the new active vertex.

6. The edge joining the vertices labelled 1 and 2 is placed in a set S.

Description of Depth First search Traversal

7. In general, let n denote the label of the current active vertex in the search and suppose that not all vertices in the component of G containing n have been visited.

We proceed as follows:

- If there are unvisited vertices adjacent to n, select the first vertex on the adjacency list of n that has not been visited and label it with the next available label.
- The vertex just labelled becomes the new active vertex.
- The edge joining n and this newly labelled vertex is placed in the set S.

Description of Depth First search Traversal

- If on the other hand, all the vertices adjacent to *n* have been visited, we backtrack *(i.e)* revisit the vertex that was the active vertex before *n* was first visited, and designate this vertex as the current active vertex.

- The general step is repeated until every vertex in that component of G has been visited.

- If not all vertices of G have been visited, then a vertex not yet visited , say the first such vertex is chosen as the next active vertex and the process continues.

Description of Depth First search(DFS) Traversal

- The label assigned to a vertex v in a graph G by the depth first search is called the **depth first search index** of *v* and is denoted by *dfi(v)*.

- When the DFS of G is completed, the number *dfi(v)* is the order in which *v* was first visited during the search.

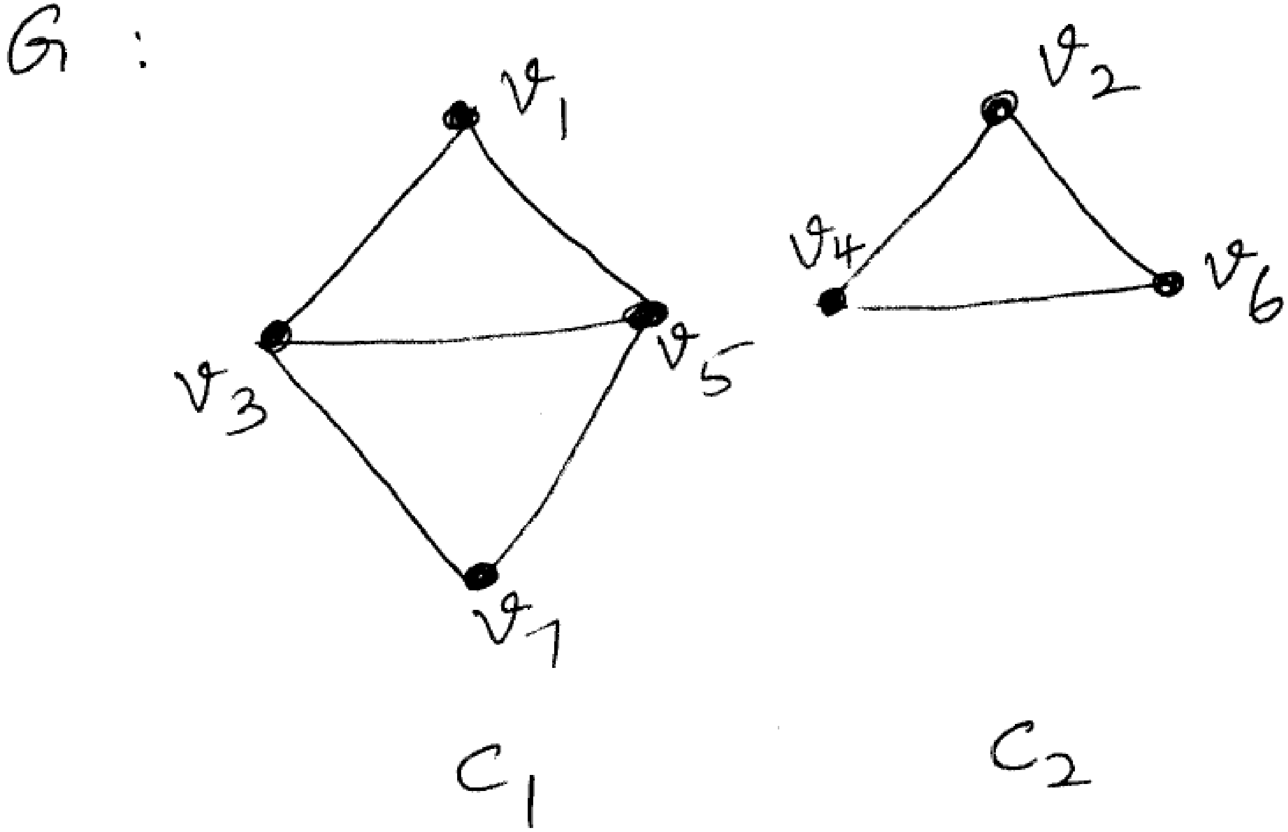
- Since each edge of G in S joins two vertices, one of which is being visited for the first time $< S >$ is a spanning forest of G, called the **Depth First Search Forest**.

Description of Depth First search(DFS) Traversal

- If G is connected, then $< S >$ is a spanning tree, called a **depth first search tree.**
- If F is a depth first search forest of a graph G, then each component of F is a rooted tree in which the root is the first vertex visited in the component.
- Each edge of G that is not an edge of F is called a **back edge.**
- Necessarily, each back edge joins two vertices in a component of G and thus in a component of F.

Example

1. Find the depth first search tree or forest in the graph G.



Solution

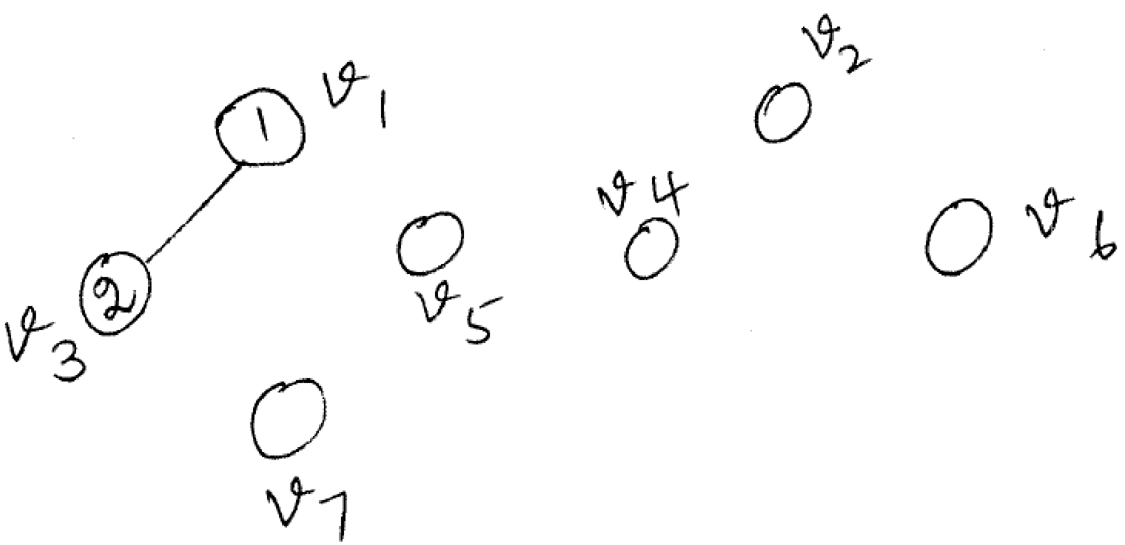
G is a disconnected graph with 2 components C_1 and C_2.

Step 1

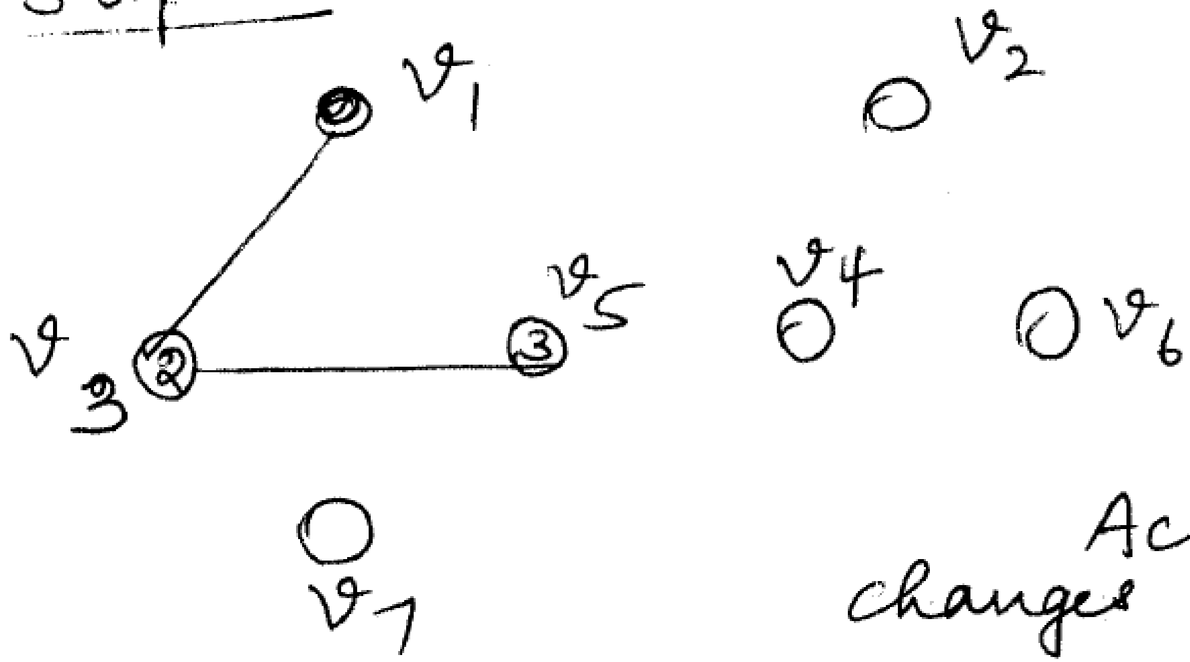
Edges in S and labels of vertices

Active vertex changes

From	To
v_1	v_3

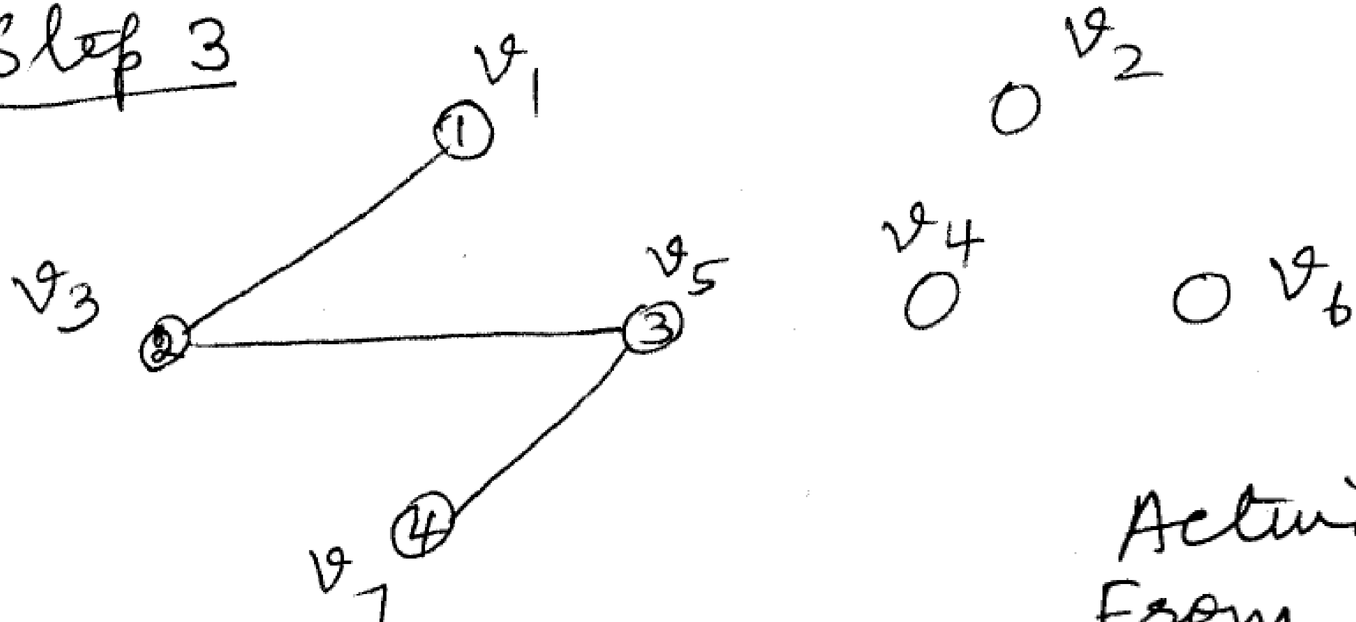


Step 2



Active vertex changes From v_3 To v_5

Step 3

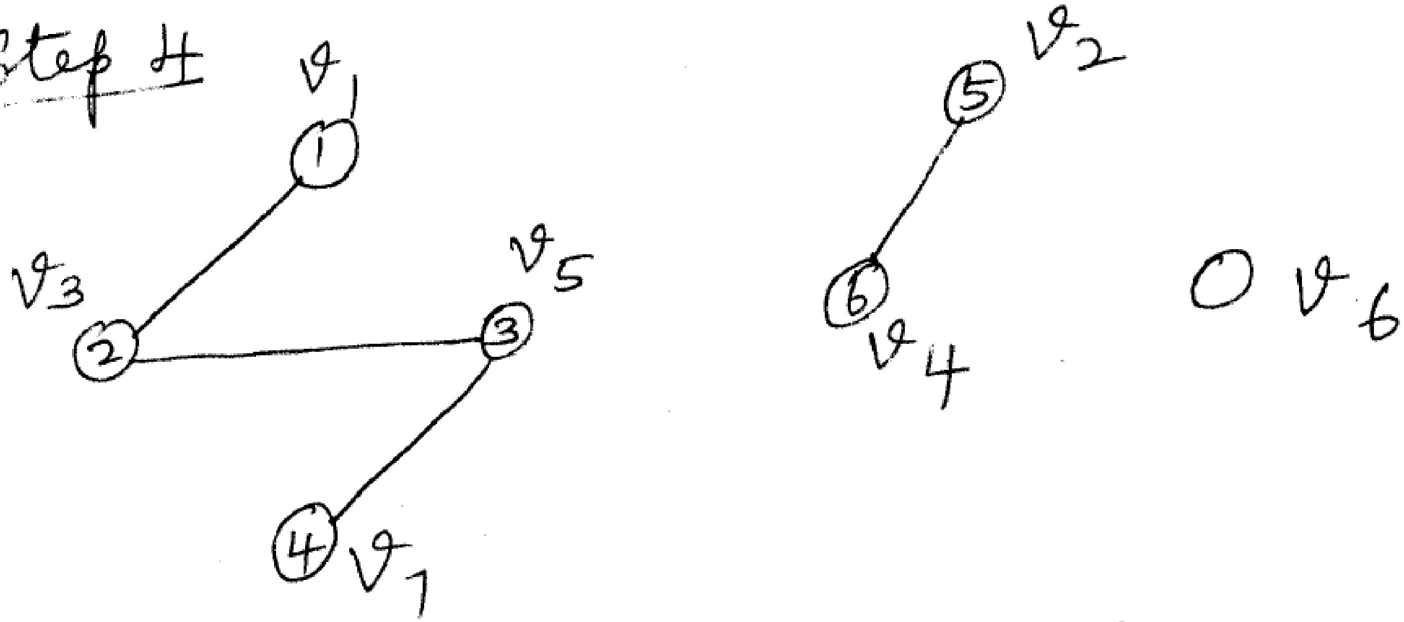


Active vertex

From	To
v_5	v_7

- All vertices in component C_1 have been visited;
- v_2 is the new active vertex.

Step 4

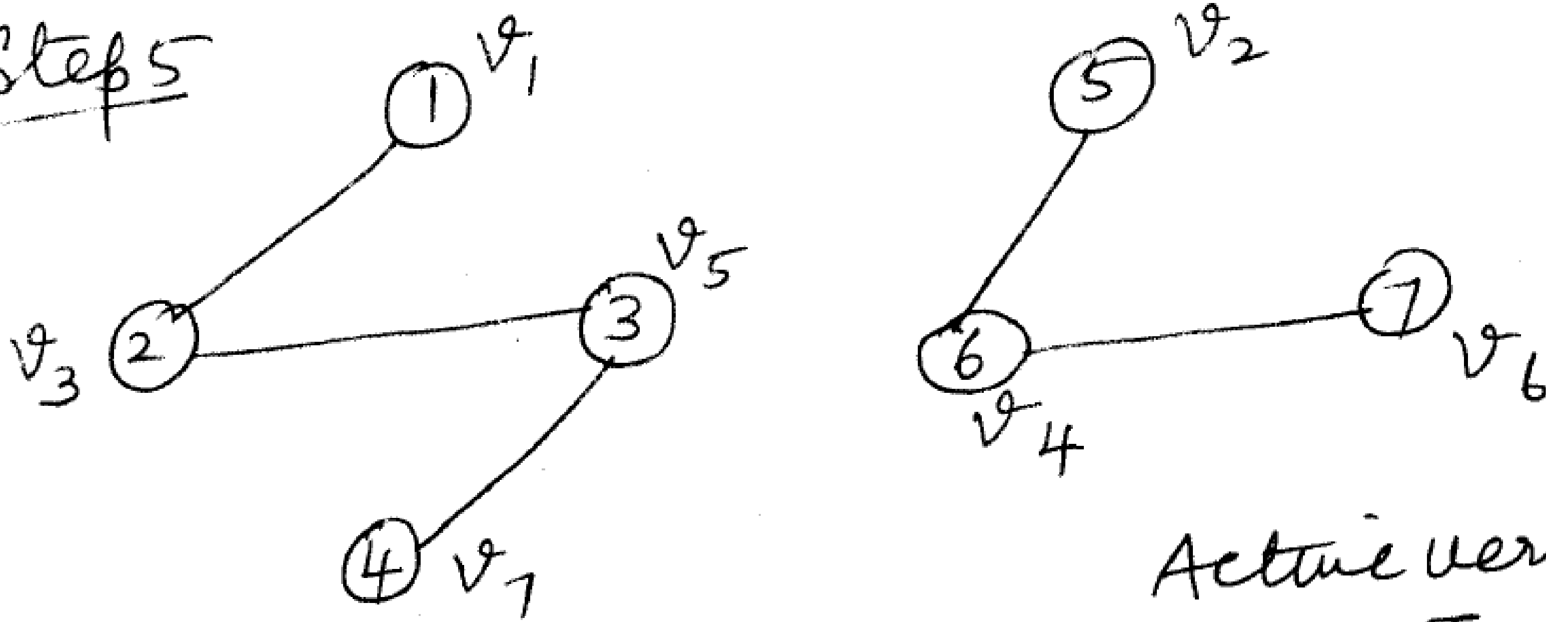


Active vertex

From To

v_2 v_4

Step 5



Active vertex

From	To
v_4	v_6

- This is the DFS forest *F*.

Depth First search(DFS) Algorithm

To conduct a depth first search of a graph G represented by its adjacency lists.

1. *dfi(v)* <- 0 for each v ε V(G)
   [Initially, all vertices are given a depth first search index of 0]

2. i <- 1
   [The parameter i is initialised and will be assigned to the i^{th} vertex visited during the search.

3. S <- ϕ
   [The set S is initialised and will be the arc set of the DFS forest.]

Depth First search(DFS) Algorithm

4. If $dfi(r) \neq 0$ for all r ε V(G), then output S and stop;
   [If not all vertices of G have been visited, a new root is selected from which a depth first search of the component of G containing that vertex is conducted.]

   Otherwise, let r be the first vertex such that $dfi(r) = 0$ and let w <- r.

5. $dfi(w)$ <- i

6. i <- $i + 1$

Depth First search(DFS) Algorithm

7. [A search is conducted for a vertex not yet visited.]

   7.1 If *dfi(v)* = *0* for some vertex *v* in the adjacency list of *w*, then continue; Otherwise go to step *7.4*.

   7.2 *S* <- *S* ∪ *{(w,v)}* and assign *Parent(v)* <- *w*

   7.3 *w* <- *v* and return to step 5.

   7.4 *If w* ≠ *r*, then *w* <- *Parent(w)* and return to step 7.1; Otherwise, return to step 4.

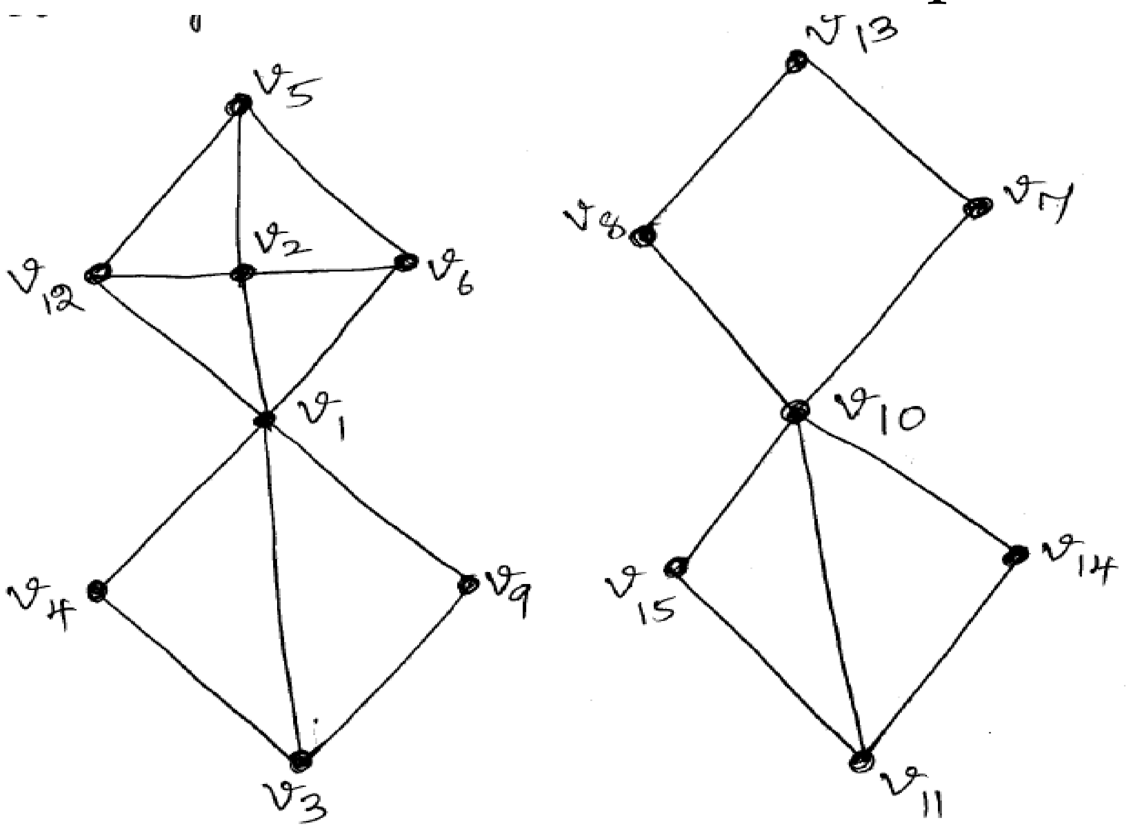
Depth First search(DFS) Algorithm

Time complexity:

If G is a graph with $|V|$ vertices and $|E|$ edges, then the complexity of a depth first search of G is $\Theta(|V| + |E|)$.

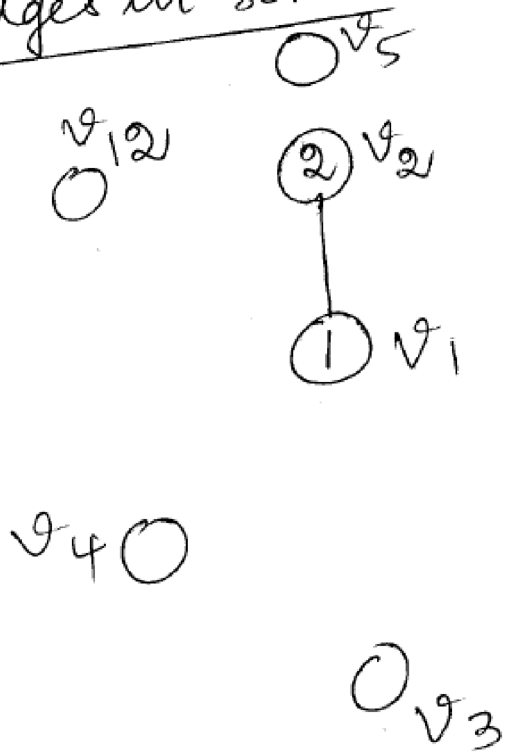
• Example

• Apply a DFS to a graph G and make a table of the DFS index and the corresponding stack.



Step 1

Edges in set S



Active vertex

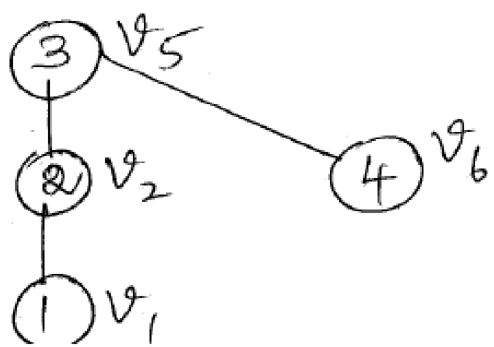
v1

Step 2

Active vertex changes from

v_1 v_2

Step 3

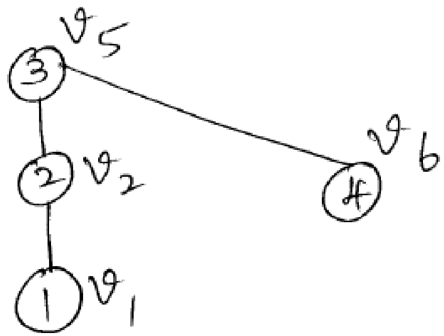


Active vertex changes from

$v_2 \rightarrow v_5$

Step 4

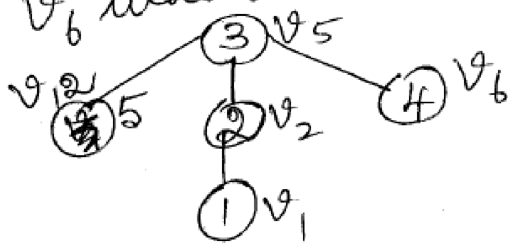
Active vertex changes from $v_5 \to v_6$



Step 5

Since all vertices adjacent to v_6 have been visited. Active vertex is the previous vertex that was the active vertex before v_6 was visited.

Active vertex $v_6 \to v_5$



Step 6 since all vertices adjacent to v_{12} have been visited, we back track & go to the previous active vertex.

Active vertex changes from

v_{12} v_5

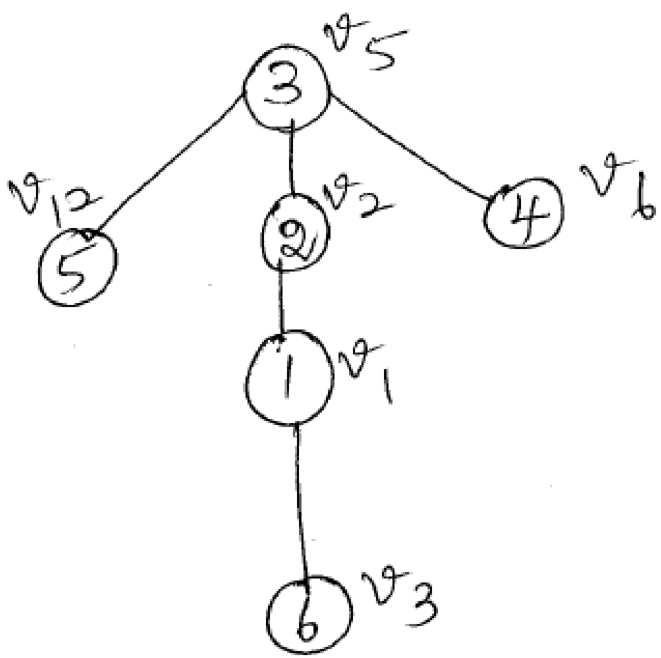
Step 7 same reason as step 6.

Active vertex changes from

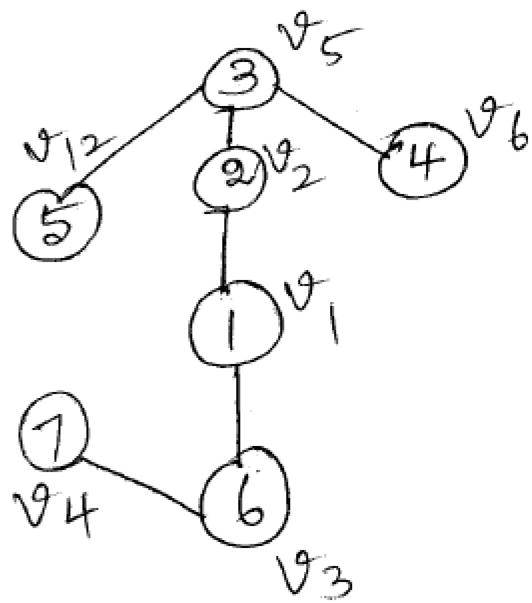
$v_5 \rightarrow v_2$

step 7 Active vertex change from v_2 to v_1.

Active vertex $v_2 \rightarrow v_1$



Step 8



Active vertex changes from $v_1 \longrightarrow v_3$

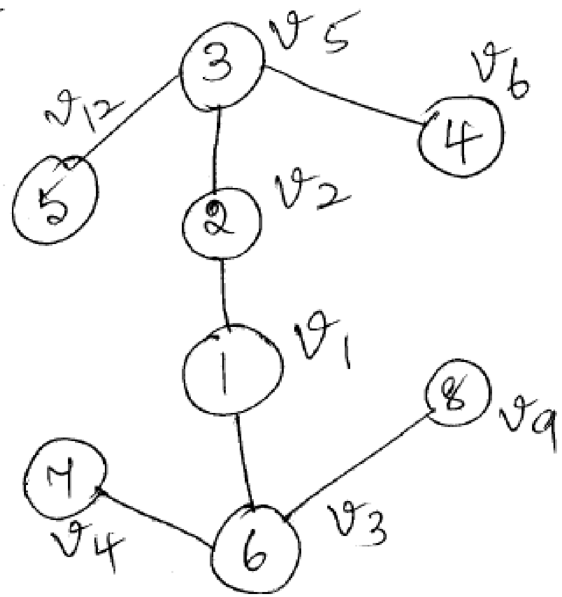
step 4

since all vertices adjacent to v_4 have been visited we back track & go to the vertex which was the active vertex before v_4.

Active vertex changes from $v_3 \rightarrow v_4$

Step 10

F_1 :



Active node changes

$v_4 \rightarrow v_3$

Step 11 since all vertices adjacent to v_9 have been visited all all vertices in this component of G have nonzero labels, we move to the next component.

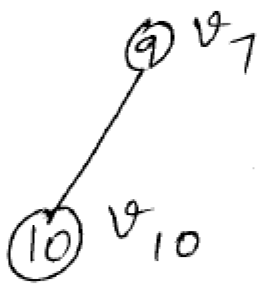
Active vertex changes from $v_3 \to v_9$

Step 12.

Active vertex changes from $v_9 \to v_7$.

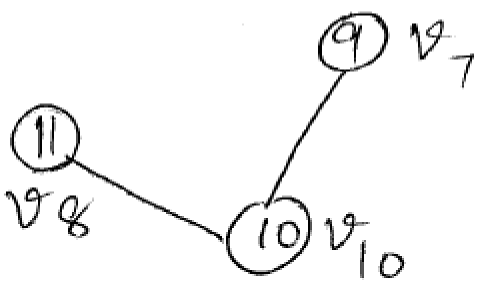
F_1 ⑨ v_7

Step 13



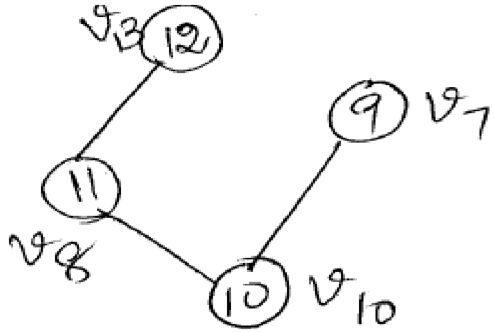
Active vertex changes from $v_7 \longrightarrow v_{10}$

Step 14



Active vertex changes from $v_{10} \longrightarrow v_8$

Step 15

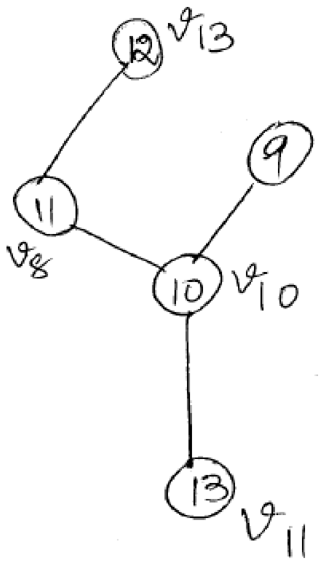


Active vertex changes from $v_8 \to v_{13}$.

Step 16. Since all vertices adjacent to v_{13} have been visited, we go to the previous active vertex.
(ie) active vertex changes from
$v_{13} \to v_8$

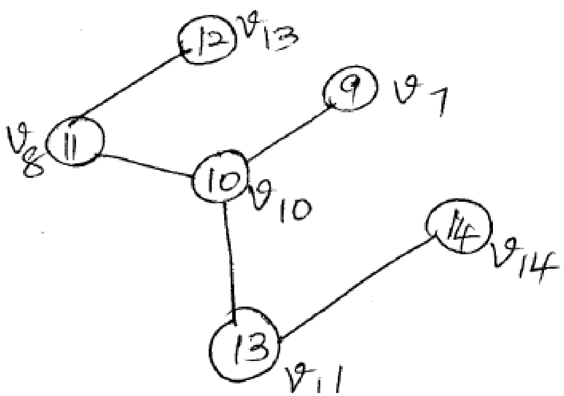
Step 17 Same as step 16. We backtrack active vertex changes from
$v_8 \to v_{10}$.

Step 18.



Active vertex changes from $v_{10} \rightarrow v_{11}$

Step 19



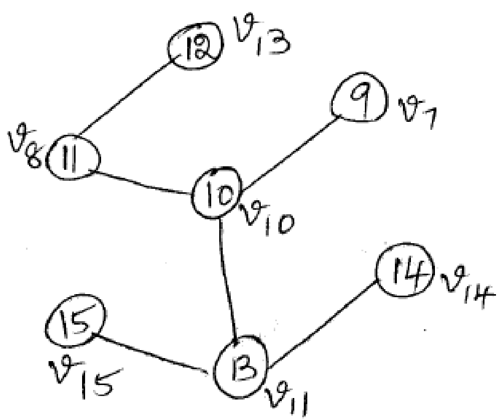
Active vertex changes from $v_{11} \rightarrow v_{14}$

Step 20 Since all vertices adjacent to v_{14} have been visited we back track to the previous active vertex.

$\therefore$ Active vertex changes from

$v_{14} \rightarrow v_{11}$

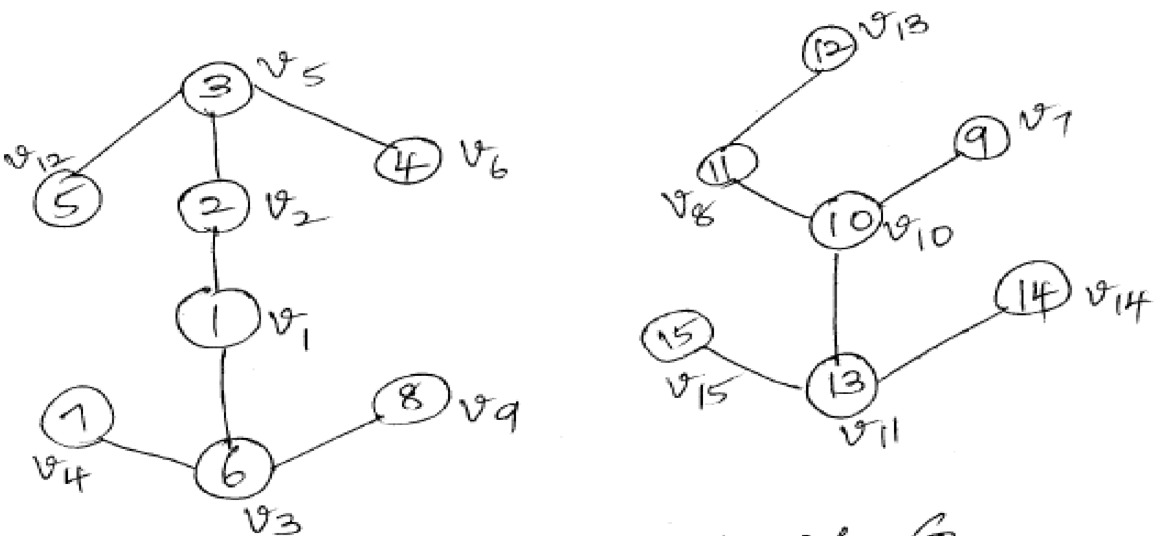
Step 21



Active vertex changes from

$v_{11} \rightarrow v_{15}$

Step 22 Since all vertices adjacent to v_{15} have been visited & by backtracking we find that all vertices have been visited (have nonzero labels), then DFS is complete & the DFS forest is output.

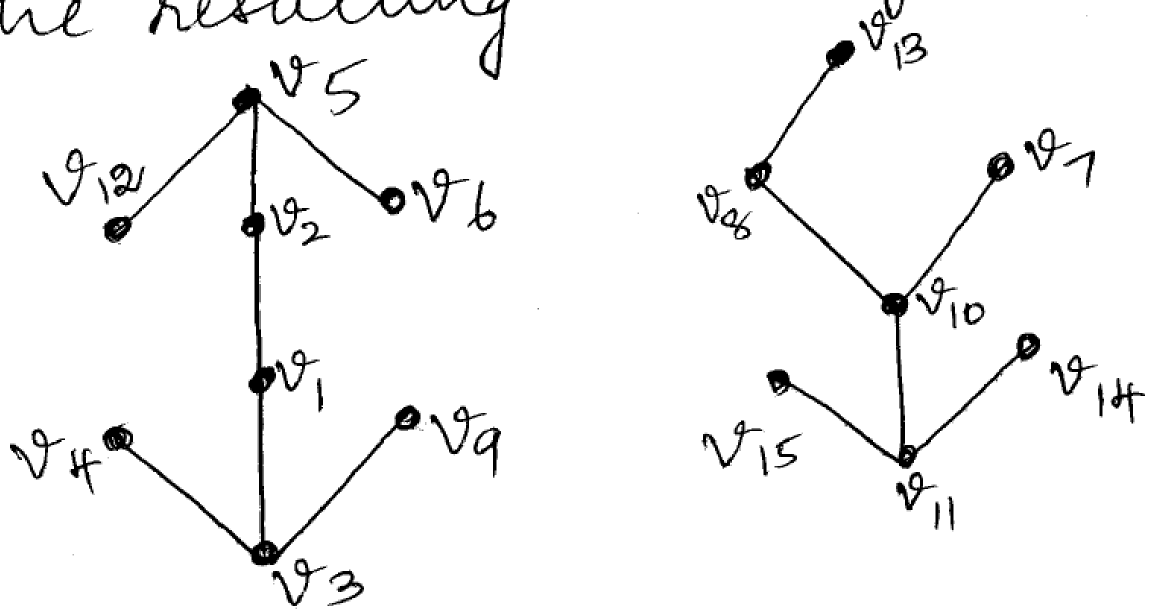


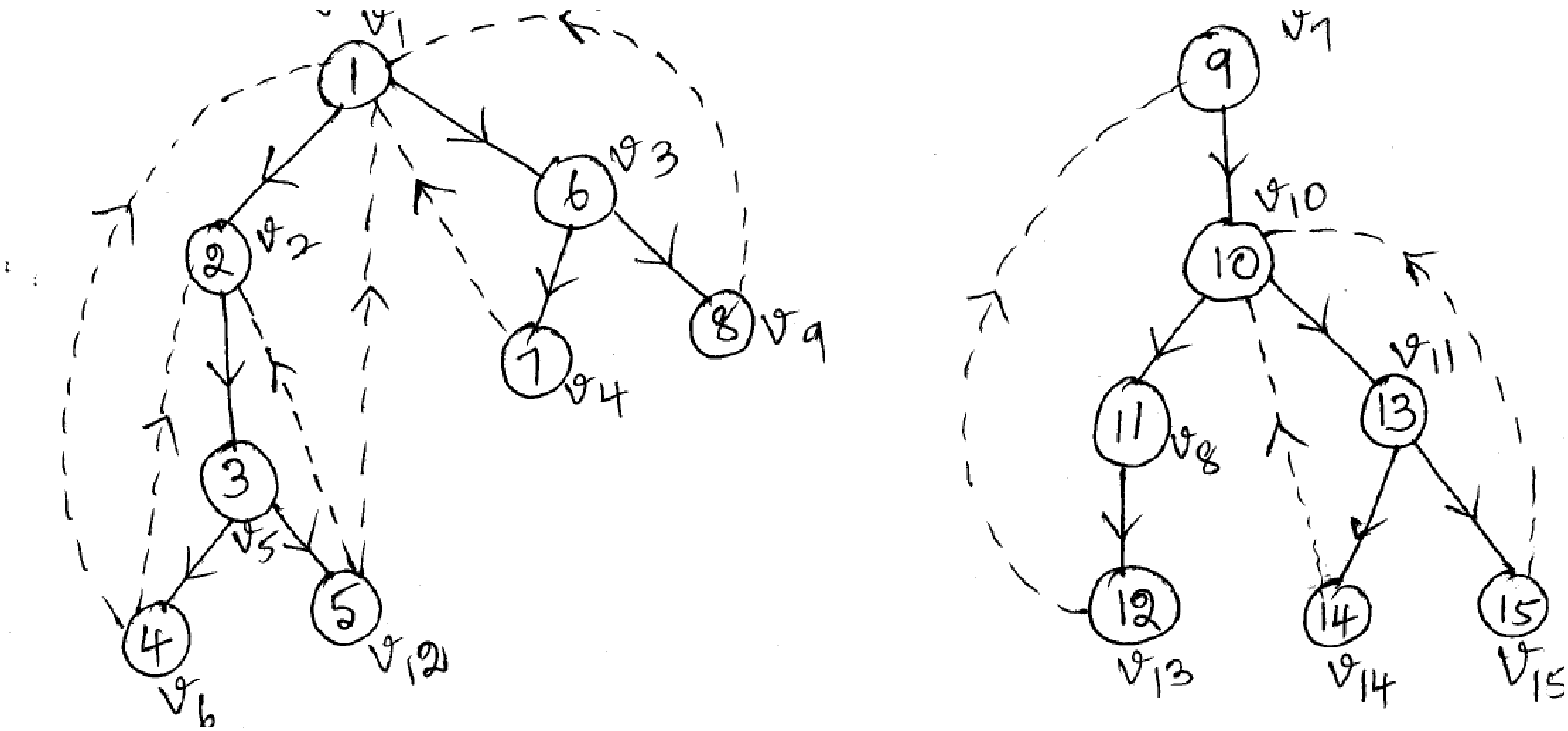
Depth first search forest of G.

G is a disconnected graph with 2 components.

The resulting DFS forest is

F :





The back edges of G are shown in the below figure by dotted lines.

v	$dfi(v)$
v_1	1
v_2	2
v_3	6
v_4	7
v_5	3
v_6	4
v_7	9
v_8	11
v_9	8
v_{10}	10
v_{11}	13
v_{12}	5
v_{13}	12
v_{14}	14
v_{15}	15

v_{15}
v_{14}
v_{11}
v_{13}
v_8
v_{10}
v_7
v_9
v_4
v_3
v_{12}
v_6
v_5
v_2
v_1

stack

Applications of Depth First Search

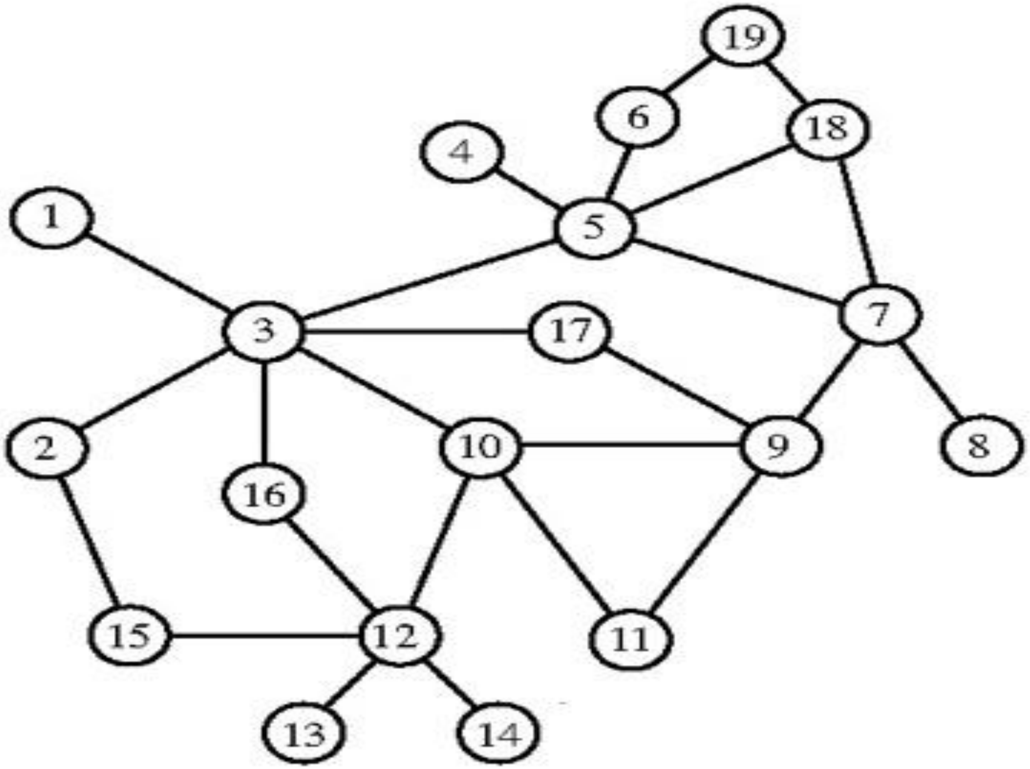
- To check if a graph is connected or disconnected.

- Finding the components of a disconnected graph

- Other applications include finding cut vertices, bridges and blocks in a graph.

Example:

- Apply a depth first search to find the DFS tree. Also, write down the stack formed and the table of depth first search index for all vertices.



- Breadth First Search

Breadth First Search (BFS)

- Another useful tool- searching technique for a graph

•The BFS visits systematically the vertices of a graph or digraph beginning at some vertex *r* of *G* also called a root.

•The root is the first active vertex.

•At any stage during the search, all the vertices adjacent from the current active vertex are scanned for vertices that have not yet been visited, that is a "broad" search is performed for unvisited vertices.

Breadth First Search (BFS)

- Each time a vertex is visited for the first time, it is labelled and added to a back of a queue.

- Note that a queue is used rather than a stack.

- The current active vertex is the one at the front of the queue.

- As soon as its neighbours have been visited, it is deleted from the queue.

Breadth First Search (BFS)

- If the queue is empty and some vertices of the graph or digraph have not yet been visited, we select any unvisited vertex , assign it a label and add it to the queue.

- When all the vertices of the graph have been visited, the search is complete.

- Assume that *G* is a graph represented by its adjacency lists.

- Initially, all vertices of *G* are labelled 0.

Breadth First Search (BFS)

- We begin by assigning *r*, the label 1 and placing *r* on a queue *Q*.

- At the next step, we delete *r* from *Q* and scan its adjacent vertices(if such vertices exist) sequentially in the order in which they appear on the adjacency list for *r*.

- The first vertex that appears on the adjacency list for *r* is assigned the next available label, namely, 2 and this vertex is then added to the back of the *Q*.

Breadth First Search (BFS)

- We continue to label the vertices adjacent to *r* and add them to *Q* until the last vertex adjacent with *r* is labelled *deg(r) + 1* and is added to *Q*.

- We then delete the next vertex from the front of the *Q*, say *w* and scan its adjacent vertices in the order in which they appear on the adjacency list of *w*.

- If a vertex adjacent with *w* still has label 0, then we assign it the next available label and add it to *Q*; Otherwise, we do not change its label.

Breadth First Search (BFS)

- We continue in this manner until Q is empty.

•If all the vertices of G are labelled with a positive integer, we stop.

•Suppose G still contains vertices labelled 0 which will happen if G is disconnected.

•Then, we select such a vertex, assign it the next available label and we continue as before.

Breadth First Search (BFS)

- This breadth first algorithm terminates once every vertex has been assigned a positive integer label.

- This algorithm actually determines a spanning forest F of G called a **breadth first search forest** where each component of F is a rooted tree.

- The root of a component of F is then the vertex having the smallest label in that component.

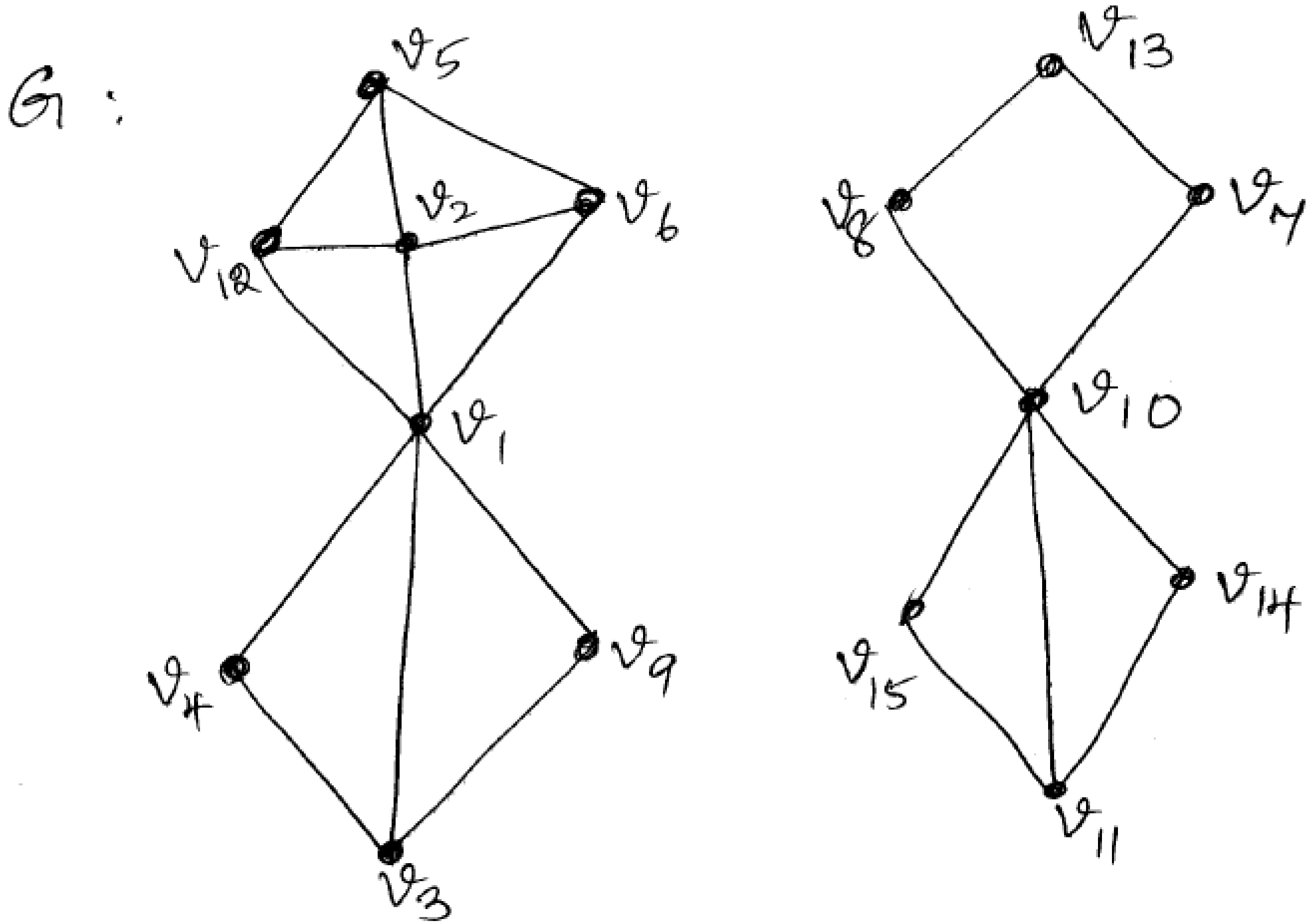
Breadth First Search (BFS)

- Further, an edge vw of G is added to F if either v is deleted from Q and w still has label 0, or w is deleted and v still has label 0.

- Time complexity of a breadth first search is $\Theta(|V|+|E|)$.

Example:

- Apply a breadth first search to the graph G.



Example:

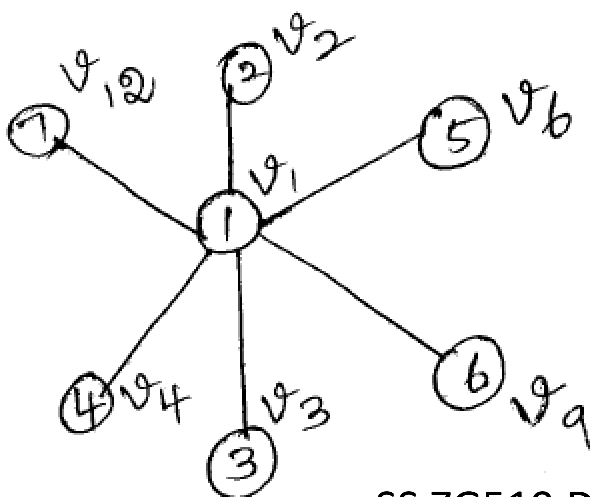
Constructing a BFS forest

Step 1

Queue

v_1

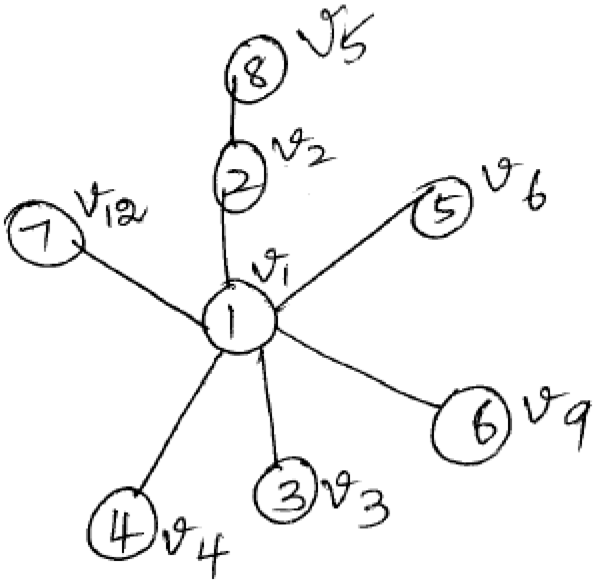
Step 2



$\cancel{v_1}$ v_2 v_3 v_4 v_6 v_9 v_{12}

Example:

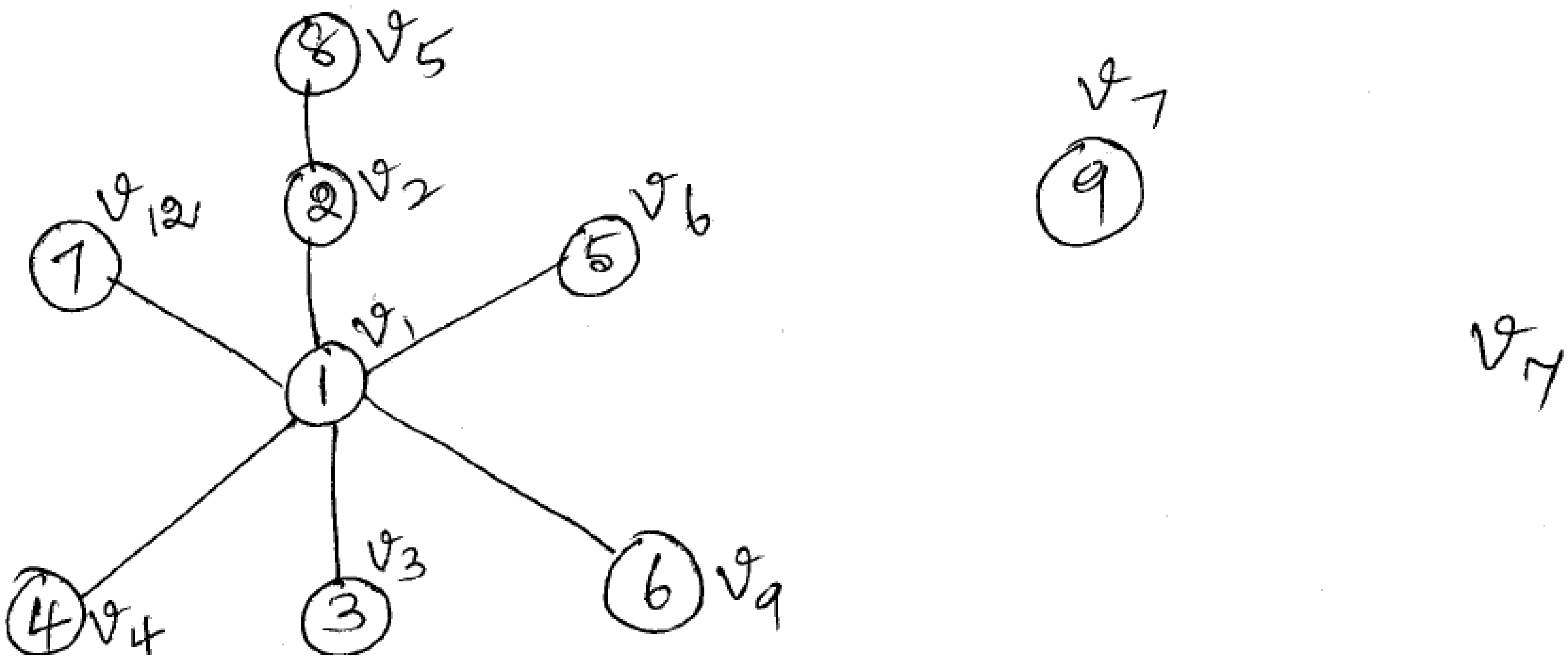
Step 3



$\cancel{v}_2\ \cancel{v}_3\ \cancel{v}_4\ \cancel{v}_6\ \cancel{v}_9\ \cancel{v}_{12}\ v_5$

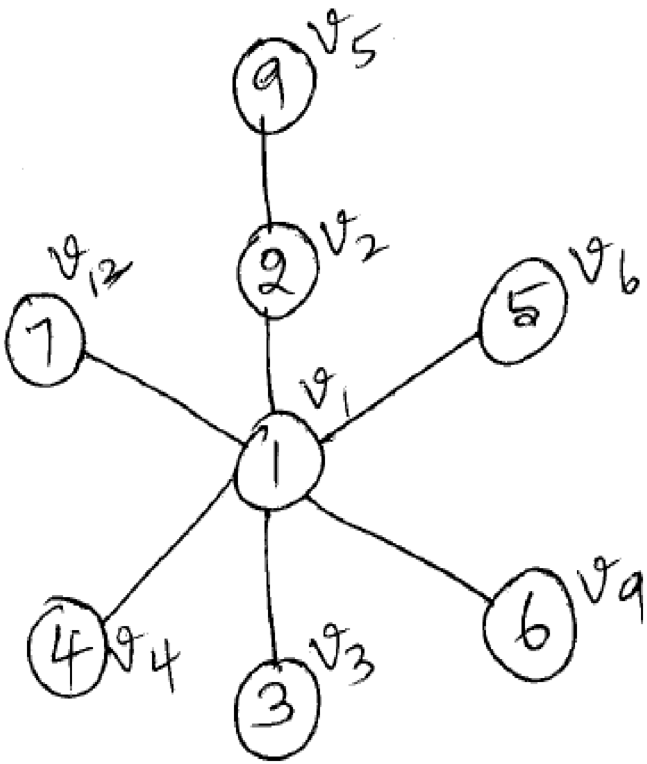
Example:

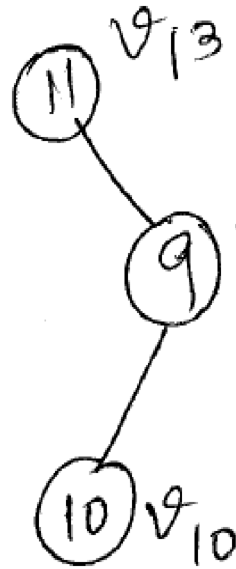
Step 4



Example:

Step 5

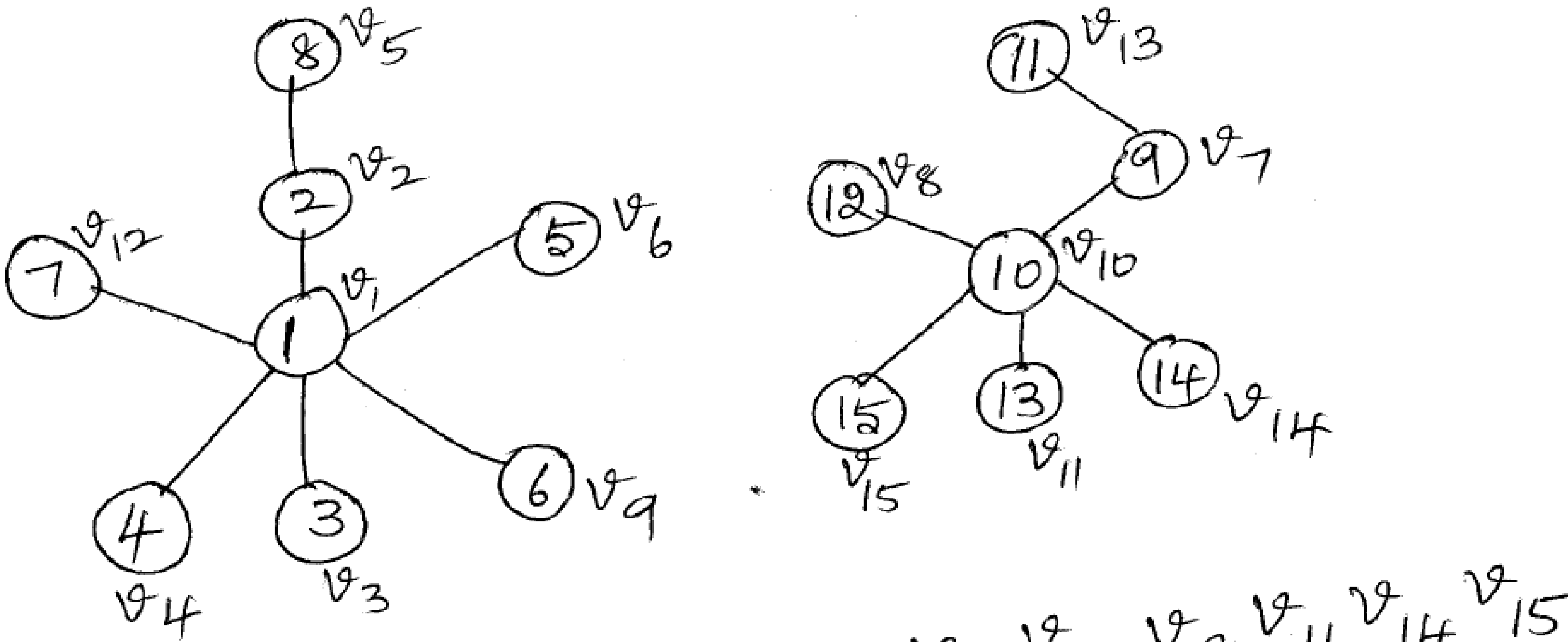




$\cancel{v_7}$ v_{10} v_{13}

Breadth First search forest of G

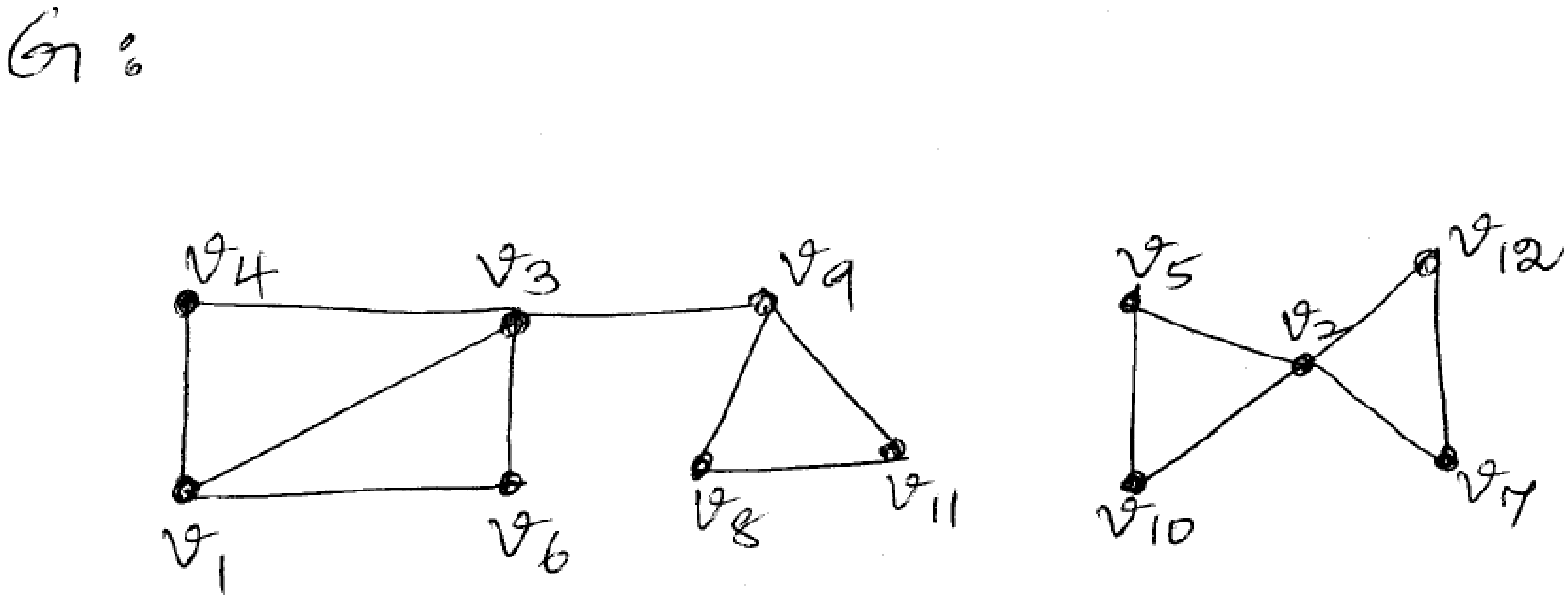
Step 6



$v_{10}\ v_{13}\ v_8\ v_{11}\ v_{14}\ v_{15}$

Example

- Apply a breadth first search to the graph G.



Example

Step 1

constructing a BFS forest

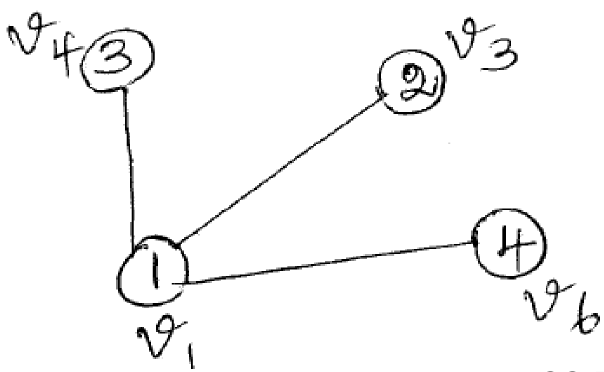
Queue

v_1

Queue

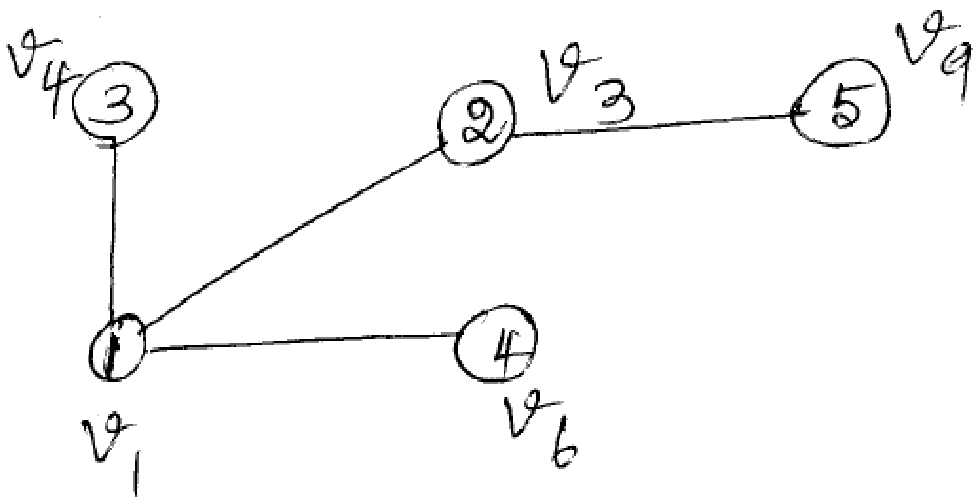
Step 2

$\cancel{v_1}$ v_3 v_4 v_6



Example

Step 3



Queue

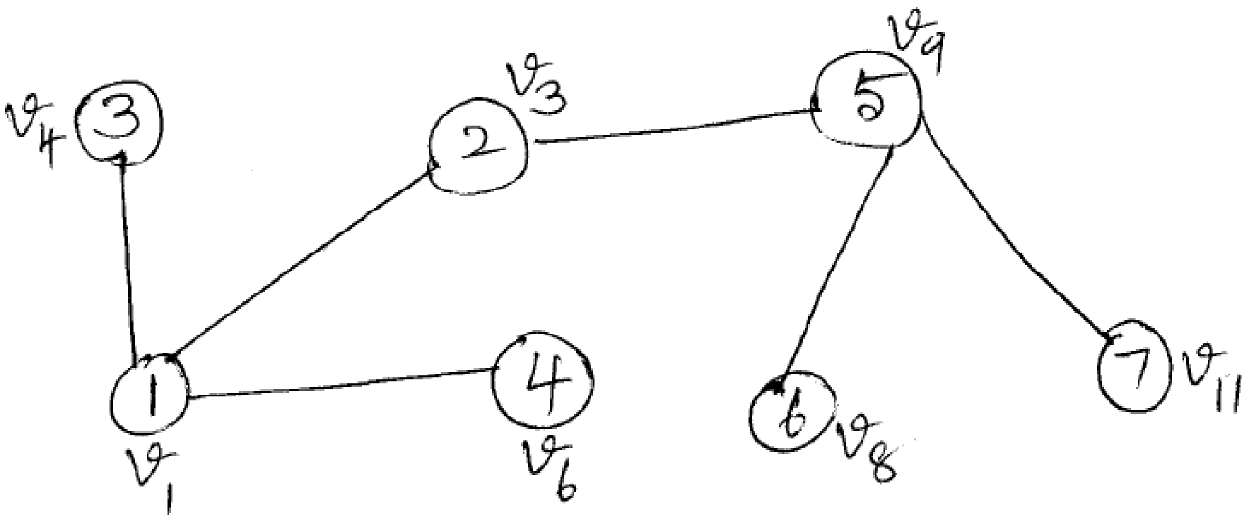
$\cancel{v_3}$ v_4 v_6 v_9

Example

Step 4

Queue

$\cancel{v_4}\ \cancel{v_6}\ \cancel{v_9}\ \cancel{v_8}\ \cancel{v_{11}}$



F_1

Example

step 5

F I

8 v_2

Queue

v_2