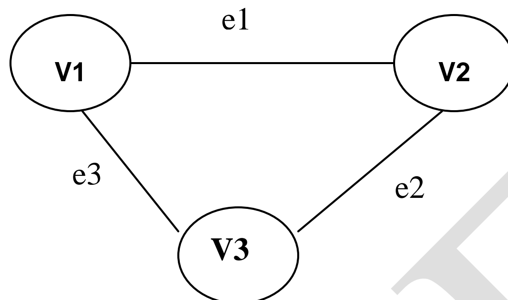
UNIT IV NON LINEAR DATA STRUCTURES – GRAPHS

Graph

A *graph* 'G = (V, E)' consist of set of *vertices* and a set of lines joining the nodes or vertices called *edges*

E.g.: -



G = (V, E)
V = {v1, v2, v3}
E = {e1, e2, e3}

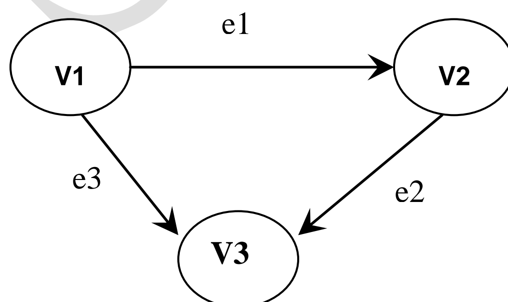
Adjacent nodes

If an edge x ∈ E for a pair of nodes (u, v), then the nodes u and v are adjacent to each other.

For the above example the nodes v1 and v2 are adjacent to each other since they are connected by the edge e1.

Directed Edge & Directed Graph or Digraph

In a Graph G =(V, E) an edge is *directed* if there is a directed arrow from one node to other node. In a graph if all the edges are directed, then that graph is said to be *directed graph* or *digraph*

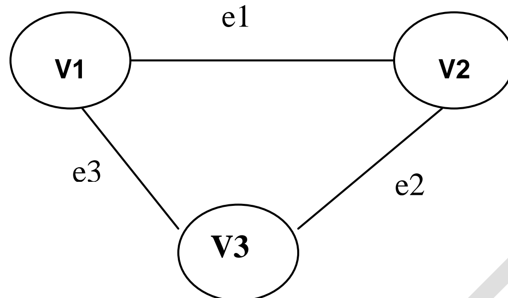


SVCET

In the above example all the edges are directed since each edge flows from one node to other node & this graph is known as directed graph
Eg A city map showing only the one way streets

Undirected Graph

In a graph if all the edges are undirected, then that graph is said to be un*directed graph*
E.g.: -

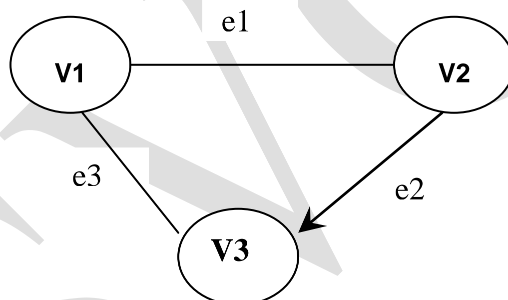


In the above example all the edges are undirected, so this is said to be undirected graph.
Eg A city map showing only the two way streets

Mixed Graph

In a graph if some edges are undirected and some are directed means such a graph is called as *mixed graph*

E.g.: -

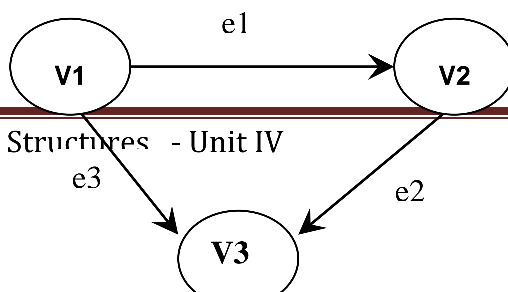


In the above example the edge e1 and e3 are undirected and e2 is directed so this is called as mixed graph.
Eg A city map showing the one way and two way streets

Initiating (or) Orginating & Terminal (or) Ending Edges

Let G = (V, E) be a graph, in this if the edge x ∈ E flow from node u to node v, then the edge x is initiating edge for node u and terminating edge for node v.

E.g.: -



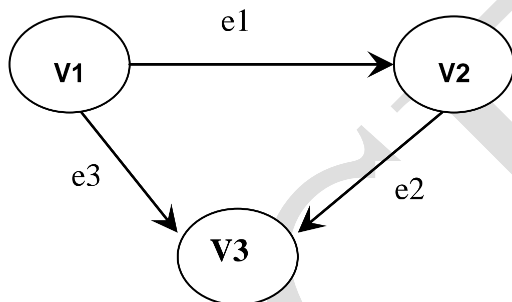
SVCET

In the above example the edge e1 is initiating edge for node v1 and terminating edge for node v2, since it flows from node v1 to node v2.

Initial & Terminal node

Let G = (V, E) be a graph, in this if the edge x ∈ E flow from node u to node v, then the node u is initial and node v is terminal node for the edge x.

E.g.: -

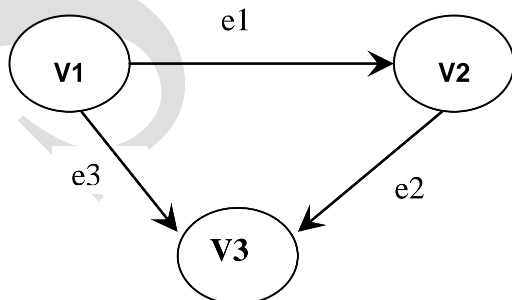


In the above example the node v1 is initial and node v2 is the terminal node for the edge e1, since it flows from node v1 to node v2.

Incident

An edge x ∈ E, which join node (u, v) whether directed or not, then x is incident to node u & v.

E.g.: -



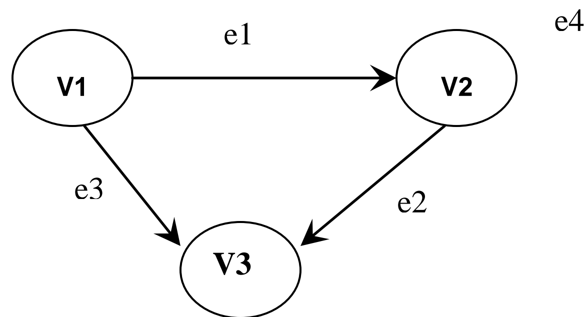
In this above example the edge e1 is incident to node v1 and v2.

Loop or sling

An edge of a graph which joins a node to itself is called is called as loop.

SVCET

E.g.: -

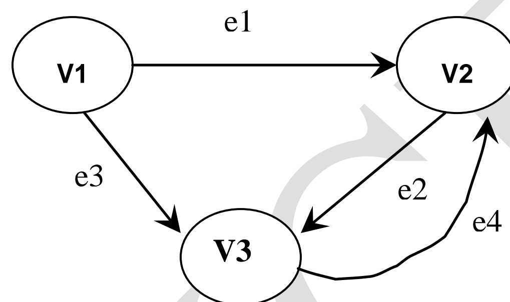


The edge e4 is loop.

Parallel Edge

The two possible edges between a pair of nodes are known as parallel edge.

E.g.: -

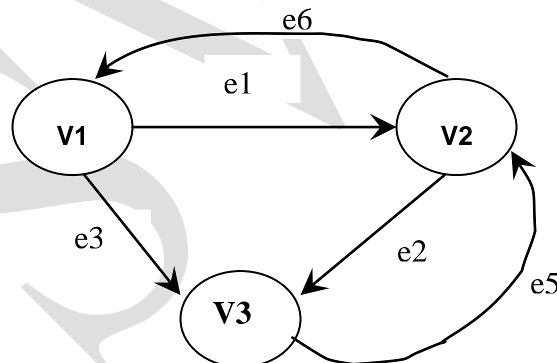


In this above example the edge e2 and e4 are parallel edges.

Multi Graph

Any graph, which has some parallel edges, is called as multigraph.

E.g.: -



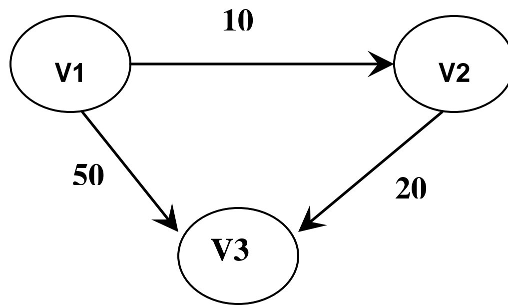
Simple Graph

If there is no more than one edge between any pair of nodes, then such a graph is called as simple graph.

Weighted Graph

A Graph in which some weights are assigned to every edge is called as weighted edge.

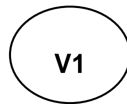
E.g.: -



Isolated Node

In a graph, a node, which is not adjacent to any other node, is called as isolated node

E.g.: -



Null Graph

A graph containing only isolated node is called as null graph.

Out Degree

For any node V, the number of edges leaving out of a node is called as out degree

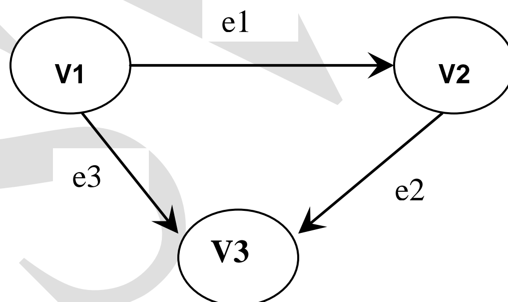
In Degree

For any node V, the number of edges entering in to a node is called as in degree

Total Degree (or) Degree of a graph

For a node the total degree is the sum of the in degree and the out degree.

E.g.: -



In the above example the degree for the nodes is

Node	In degree	Out degree	Total Degree
V1	0	1	1
V2	1	1	2
V3	2	0	2

Note

Degree for a loop = 2
Degree for an isolated node = 0

Path

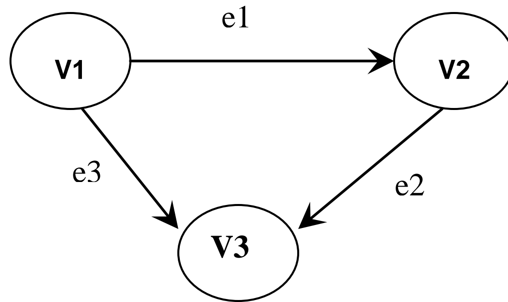
Path is defined as sequence of edges in a graph

(V1, v2, v3, …, vn)

Path is to traverse through the nodes appearing in the sequence originating in the initial node first edge and end at the terminal node of the last edge in the sequence. The length of the path is the number of nodes appearing in the path.

SVCET

E.g.: -



The path is **e1, e2.** The length is **2.**

Simple path

If the edges present in the path are distinct, then such a path is called as simple path.

Elementary path

If the nodes present in the path are distinct, then the path is said to be elementary path.

Cycle or Circuit

If the path originates and end at the same node, then the path is said to be cycle.

SHORTEST PATH PROBLEM

We have seen in the graph traversals that we can travel through edges of the graph. It is very much likely in applications that these edges have some weights attached to it. This weight may reflect distance, time or some other quantity
that corresponds to the cost we incur when we travel through that edge. For example, in the graph in Figure 18, we can go from Delhi to Andaman Nicobar through Madras at a cost of 7 or through Calcutta at a cost of 5. (These
numbers may reflect the airfare in thousands.) In these and many other applications, we are often required to find a shortest path, i.e a path having the minimum weight between two vertices. In this section, we shall discuss this
problem of finding shortest path for directed graph in which every edge has a non-negative weight attached.

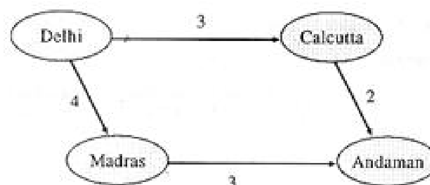


Figure 18: A graph connecting four cities

Let us at this stage recall how do we define a path. A path in a graph is sequence of vertices such that there is an edge that we can follow between each consecutive pair of vertices. Length of the path is the sum of weights of the
edges on that path. The starting vertex of the path is called the source vertex and the last vertex of the path is called the destination vertex. Shortest path from vertex v to vertex w is a path for which the sum of the weights of the arcs
or edges on the path is minimum.

Here you must note that the path that may look longer if we see the number of edges and vertices visited, at times may be actually shorter costwise.

Also we may have two kinds of problems in finding shortest path. One could be that we have a single source vertex and we seek a shortest path from this source vertex v to every other vertex of the graph. It is called single source
shortest path problem.

Consider the weighted graph in Figure 19 with 8 nodes A,B,C,D,E,F,G and H.

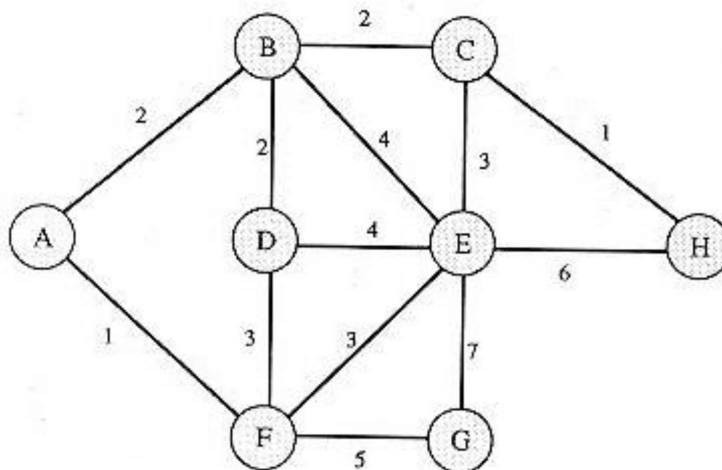


Fig. 19 : A Weighted Graph

We may further look for a path with length shorter than 13, if exists. For graphs with a small number of vertices and edges, one may exploit all the permutations combinations to find shortest path. Further we shall have to exercise this
methodology for finding shortest path from A to all the remaining vertices. Thus, this method is obviously not cost effective for even a small sized graph.

There exists an algorithm for solving this problem. It works like as explained below:

Let us consider the graph in Figure 18 once again.

1. We start with source vertex A.
2. We locate the vertex closest to it, i.e. we find a vertex from the adjacent vertices of A for which the length of the edge is minimum. ere B and F are the adjacent vertices of A and Length (AB) Length (AF)
   Therefore we choose F.
3. Now we look for all the adjacent vertices excluding the just earlier vertex of newly added vertex and the remaining adjacent vertices of earlier vertices, i.e. we have D,E and G (as adjacent vertices of F) and B (as remaining adjacent vertex of A).
4. Now we again compare the length of the paths from source vertex to these unattached vertices, i.e. compare length (AB), length (AFD), length (AFG) and length (AFE). We find the length (AB) the minimum. There we choose vertex B.
5. Breadth First Search (BFS)
6.
7. In DFS we pick on one of the adjacent vertices; visit all of its adjacent vertices and back track to visit the unvisited adjacent vertices. In BFS, we first visit all the adjacent vertices of the start vertex and then visit all -the unvisited
8. vertices adjacent to these and so on. Let us consider the same example, given in Figure l5. We start say, with v1. Its adjacent vertices are V2, V8, V3. We visit all one by one. We pick on one of these, say v2. The unvisited adjacent
9. vertices to v2 are v4, v5. We visit both. We go back to the remaining visited vertices of v1 and pick on one of those, say v3. The unvisited adjacent vertices to v3 are v6, v7. There are no more unvisited adjacent vertices of v8, v4,
10. v5, v6 and v7.

SVCET

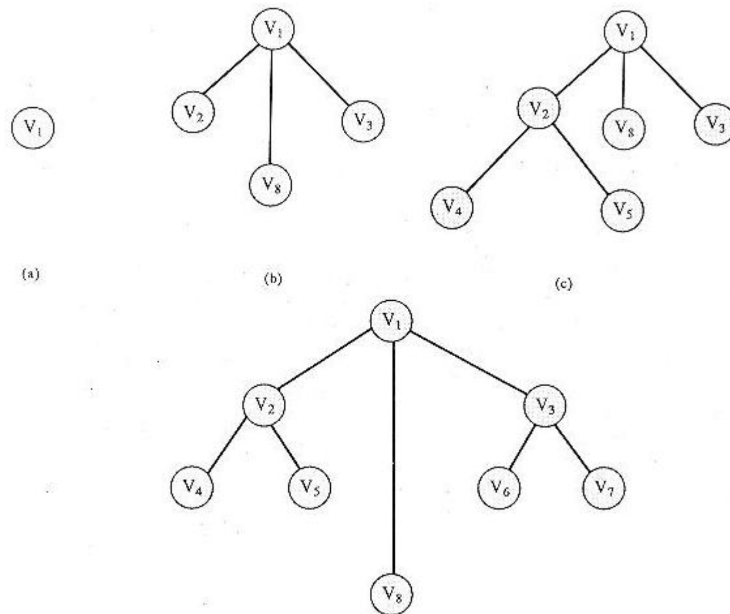


Figure 17:

11.

12. Thus, the sequence so generated is v1, v2, v8, v3, v4, v5, v6, v7. Here we need a queue instead of a stack to implement it. We add unvisited vertices adjacent to the one just visited at the rear and read at from to find the next vertex

13. to visit.

14. To implement breadth-first search, we change stack operations to queue operations in the stack-based search program above:

```
15. Queue queue(maxV);
16. void visit (int k) // BFS, adjacency lists
17. {
18. struct node *t;
19. queue.put (k);
20. while (!queue.empty()))
21. {
22.   k = queue.geto; val[k] = ++id;
23.   for (t = adj[k]; t ! = z; t = t-ncxt)
24.   if (val[t-vl == unseen)
25.     ( queue.put(t-v); val[t-vl = 1;}
```

```
26.        }
27.        }
```

SVCET

Depth First Search (DFS)

In graphs, we do not have any start vertex or any special vertex singled out to start traversal from. Therefore the traversal may start from any arbitrary vertex.

We start with say, vertex v. An adjacent vertex is selected and a Depth First Search is initiated from it, i.e. let V1, V2 .. Vk are adjacent vertices to vertex v. We may select any vertex from this list. Say, we select v1 . Now all the

adjacent vertices to v1 are identified and all of those are visited; next V2 is selected and all its adjacent vertices visited and so on. This process continues till all the vertices are visited. It is very much possible that we reach a traversed

vertex second time. Therefore we have to set a flag somewhere to check if the vertex is already visited. Let us see it through an example. Consider the following graph

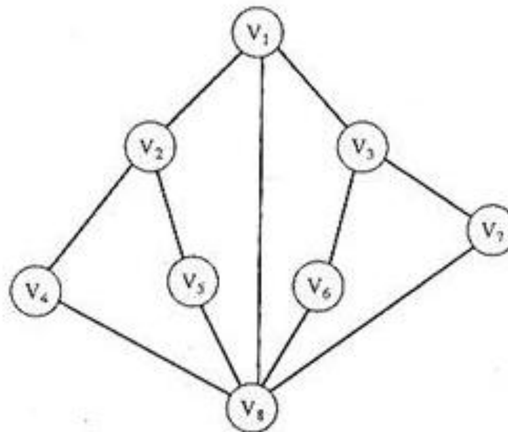


Fig 15 : Example Graph for DFS

Let us start with v1.

Its adjacent vertices are v2, v8 and V3. Let us pick on V2.

Its adjacent vertices are v1 ,v4, v5. v1 is already visited. Let us pick on V4.

Its adjacent vertices are V2, V8.

v2 is already visited. Let us visit v8. Its adjacent vertices are V4, V5, V1, V6, V7.

V4 and v1, are already visited. Let us traverse V5.

Its adjacent vertices are v2, v8. Both are already visited Therefore, we back track.

We had V6 and V7 unvisited in the list of v8. We may visit any. We visit v6.

Its adjacent are v8 and v3. Obviously the choice is v3.

Its adjacent vertices are v1, v7 . We visit v71.

All the adjacent vertices of v7 are already visited, we back track and find that we have visited all the vertices.

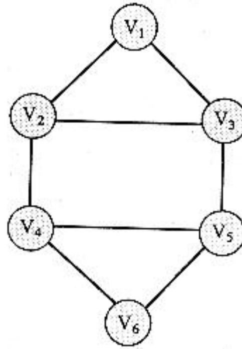


Fig. 16 : Example Graphs for DFS

Therefore the sequence of traversal is

v1,v2 ,v4 , v8 , v5 , v6 , v3 , v7.

This is not a unique or the only sequence possible using this traversal method.

Let us consider another graph as given in Figure 16.

Is v1 , v2 , v3 , v5 , v4 , v6 a traversed sequence using DFS method?

We may implement the Depth First Search method by using a stack. pushing all unvisited vertices adjacent to the one just visited and popping the stack to find the next vertex to visit.

Implementation

We use an array val[V] to record the order in which the vertices are visted. Each entry in the array is initialized to the value unseen to indicate that no vertex has yet been visted. The goal is to systematically visit all the vertices of the

graph, setting the val entry for the ith vertex visited to i, for i = 1, 2...... ,V. The following program uses a procedure visit that visits all the vertices in the same connected component as the vertex given in the argument.

void search( )

```
{
int k;
for (k = 1; k V; k++) val[k] = unseen;
for (k = 1; k V; k++)
   if (val[k] == unseen) visit(k);
}
```

The first for loop initializes the val array. Then, visit is called for the first vertex, which results in the val values for all the vertices connected to that vertex being set to values different from unseen. Then search scans through the val
array to find a vertex that hasn't been seen yet and calls visit for that vertex, continuing in this way until all vertices have been visited. Note that this method does not depend on how the graph is represented or how visit is implemented.
First we consider a recursive implementation of visit for the adjacency list representation: to visit a vertex, we check all its edges to see if they lead to vertices that have not yet been seen; if so, we visit them.

```
void visit (int k) // DFS, adjacency lists
{
struct node *t;
val[k] = ++i;
for (t = adj [k]; t ! = z; t = t-next)
   if (val[t-vl == unseen) visit (t-v);
}
```

We move to a stack-based implementation:

```
Stack stack(maxV);
void visit(int k) // non-recursive DFS, adjacency lists
{
struct node *t
stack.push(k);
while (!stack.empty ( ))
{
k = stack.pop(); val[k] = ++id;
```

SVCET

```
for (t = adj[k]; t != z; t = t-next)
if (val[t-v} == unseen)
{stack.push(t-v); val[t-v] = 1;}
```

Vertices that have ben touched but not yet visited are kept on a stack. To visit a vertex, we traverse its edges and push onto the stack any vertex that has, not yet been visited and that is not already on the stack. In the recursive
implementation, the bookkeeping for the "partially visited" vertices is hidden in the local variable t in the recursive procedure. We could implement this directly by maintaining pointers (corresponding to t) into the adjacency lists, and
so on.
Depth-first search immediately solves some basic graph- processing problems. For example, the procedure is based on finding the connected components in turn; the number of connected components is the number of times visit is
called in the last line of the program. Testing if a graph has a cycle is also a trivial modification of the above program. A graph has a cycle if and only if a node that is not unseen is discovered in visit. That is, if we encounter an edge
pointing to a vertex that we have already visited, then we have a cycle

SVCET

DO we have a path from any vertex to any other vertex in the above example? If you see it carefully, you may find the answer to the above question as YES. Such a graph is said to be connected graph. A graph is called connected if
there exists a path from any vertex to any other vertex. There are graphs which are unconnected. Consider the graph in figure 4.

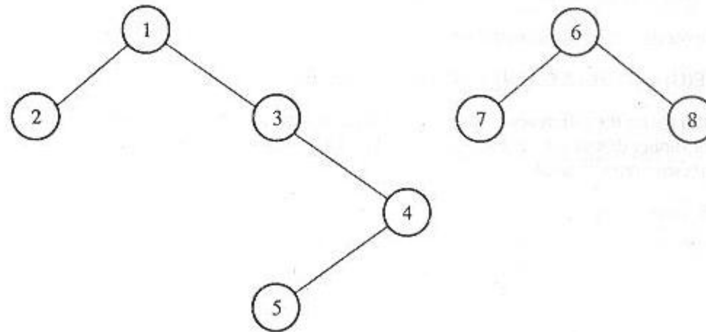


Figure 4 : An Unconnected Graph

It is an unconnected graph. You may say that these are two graphs and not one. Look at the figure in its totality and apply the definition of graph. Does it satisfy the definition of a graph? It does. Therefore, it is one graph having two
unconnected components. Since there are unconnected components, it is an unconnected graph.

So far we have talked of paths, cycles and connectivity of undirected graph. In a Digraph the path is called a directed path and a cycle as directed cycle.

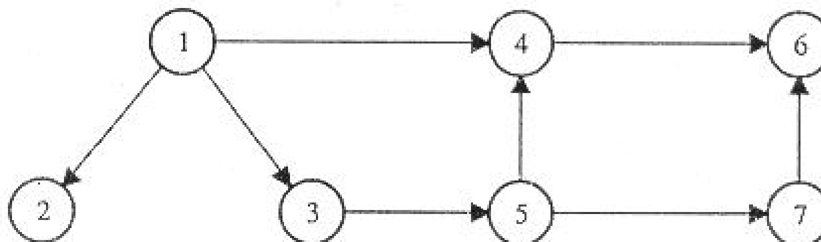


Figure 5 : A Digraph

In Figure 5 1,2 is a directed path; 1,3,5,7,6 is a directed path 1,4,5 is not a directed path.

There is no directed cycle in the above graph. You may verify the above statement. A digraph is called strongly connected if there is a directed path from any vertex to any other vertex.
Consider the digraph given in Figure 6.

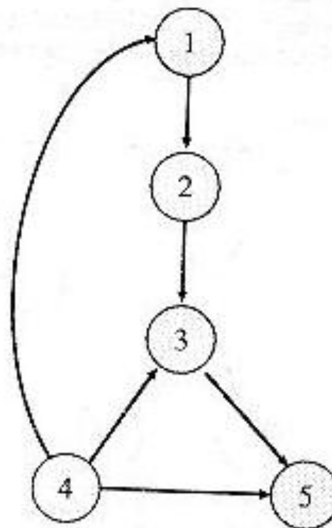


Fig. 6 : A Weakly Connected Graph