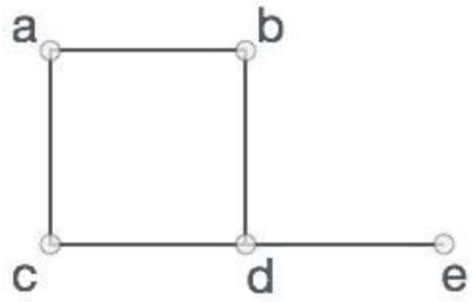
UNIT 5

Graph

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

Graph is a collection of nodes and edges in which nodes are connected with edges



In the above graph,

V = {a, b, c, d, e}

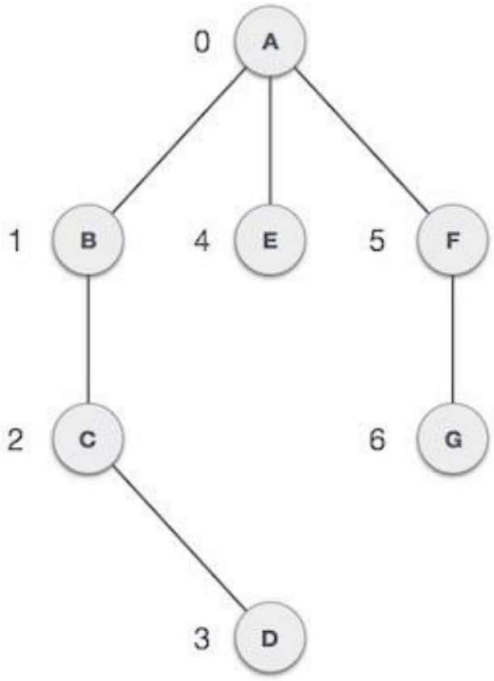
E = {ab, ac, bd, cd, de}

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



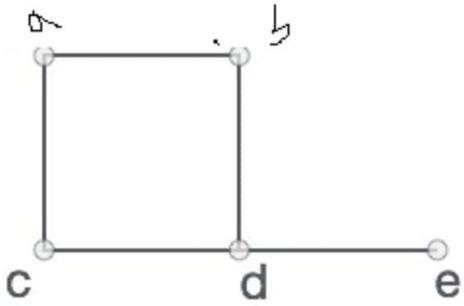
Basic Operations

Following are basic primary operations of a Graph −

- **Add Vertex** − Adds a vertex to the graph.
- **Add Edge** − Adds an edge between the two vertices of the graph.
- **Display Vertex** − Displays a vertex of the graph.

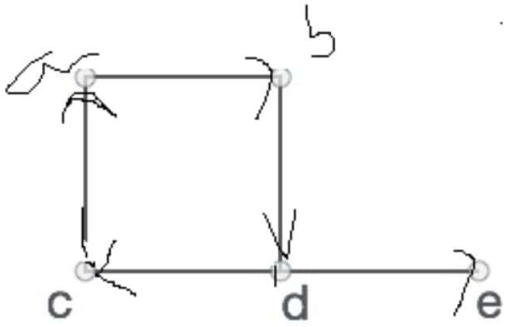
Undirected Graph

A graph with only undirected edges is said to be undirected graph.



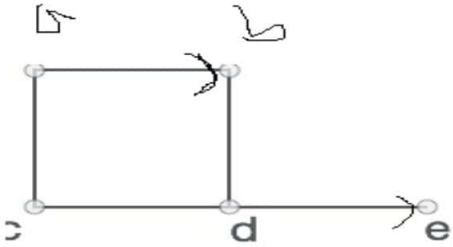
Directed Graph

A graph with only directed edges is said to be directed graph.



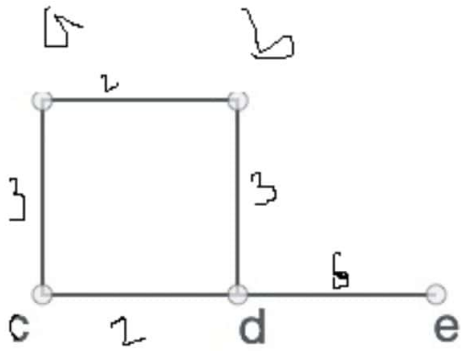
Mixed Graph

A graph with both undirected and directed edges is said to be mixed graph.



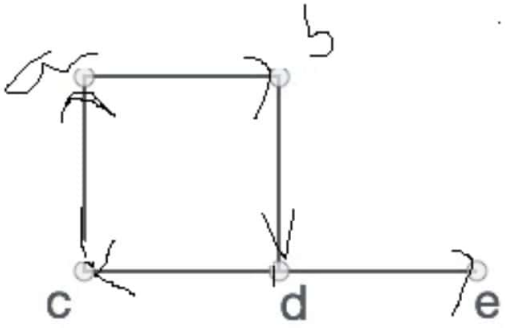
Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.



Outgoing Edge

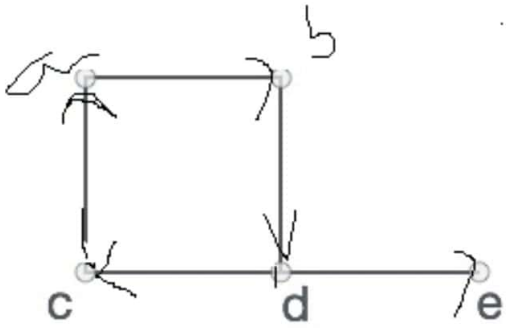
A directed edge is said to be outgoing edge on its origin vertex.



Here from a outgoing edge is **ab**

Incoming Edge

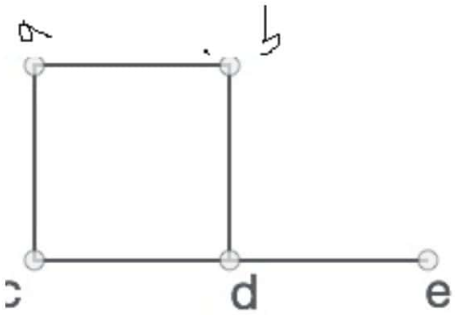
A directed edge is said to be incoming edge on its destination vertex.



Here from D: incoming edge is **BD**

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.



D degree : 3

E degree : 1

Graph degree: 5

Graph Representations:

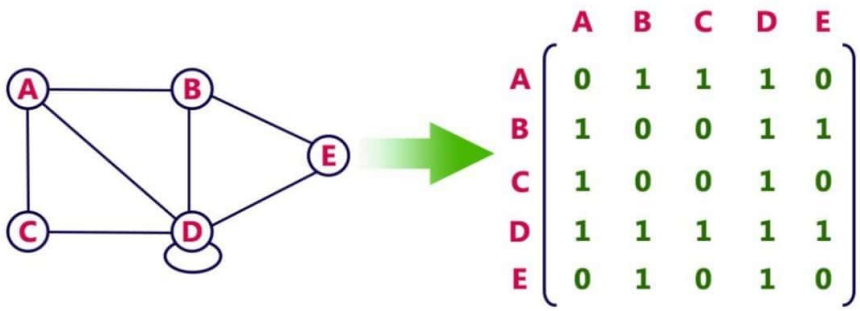
Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

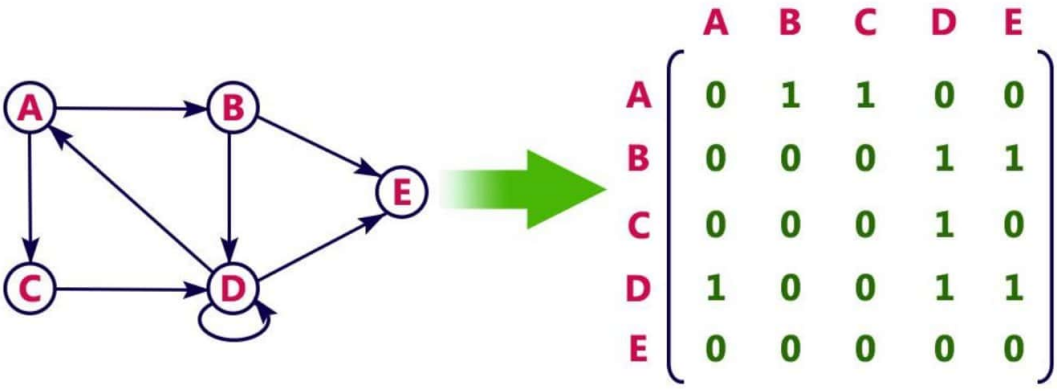
Adjacency Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



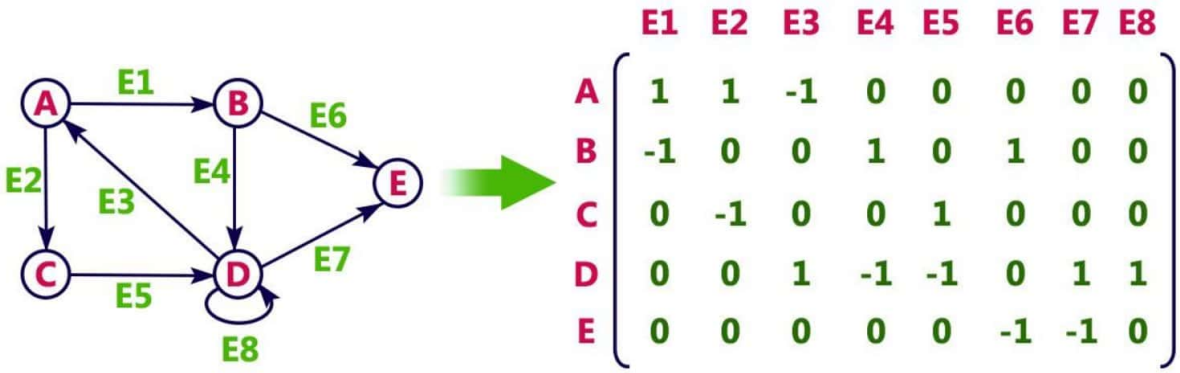
Directed graph representation...



Incidence Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

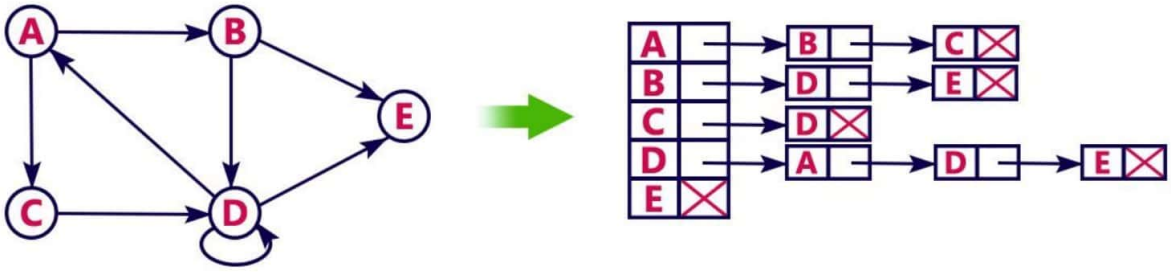
For example, consider the following directed graph representation...



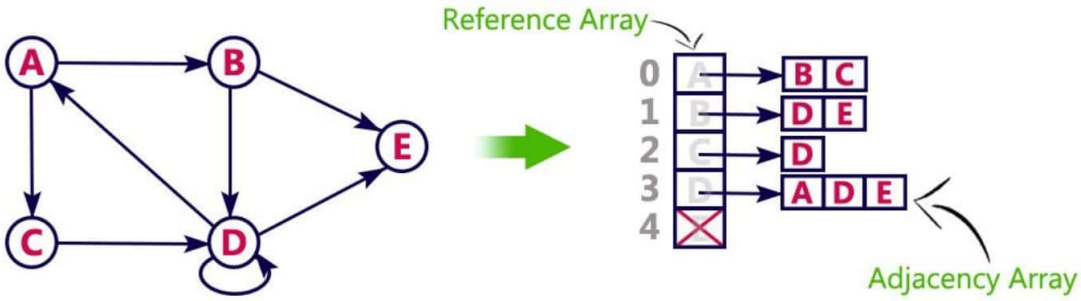
Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



Operations on Graphs:—

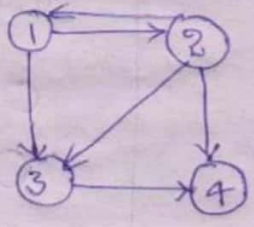
* Storing or creating graphs.
* Insert vertex.
* Delete vertex.
* Insert edge
* Delete edge
* Find vertex.

(i) Storing & creating graphs:—

There are two methods for storing.

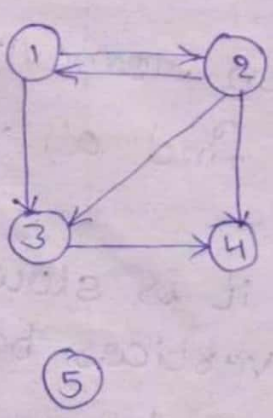
1. Adjacency list
2. Adjacency matrix.

(ii) Insert vertex:—



	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	0	0	0	1
4	0	0	0	0

```
addVertex (Vn);
```



	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	0
3	0	0	0	1	0
4	0	0	0	0	0
5	0	0	0	0	0

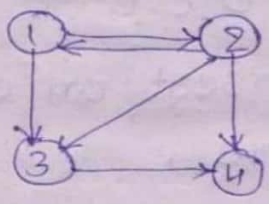
Scanned with CamScanner

This function inserted one more node into the graph, after inserting the graph size becomes increase by one. So, the size of matrix (representation of graph) increase by 1 at column level & row level.

Means., simply after inserting vertex nxn becomes (n+1) x (n+1).

The newly inserted vertex do not have indegree & outdegree.

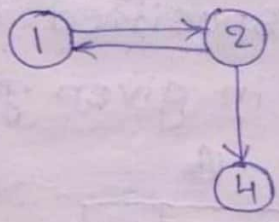
ii) Delete Vertex :—



	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	0	0	0	1
4	0	0	0	0

deleteVertex(VG);

Delete vertex 3 :—



	1	2	4
1	0	1	0
2	1	0	1
4	0	0	0

This function used to delete specified node/vertex which are present in the stored graph G.

If node is present then matrix (representation of graph) that vertex number column & row.

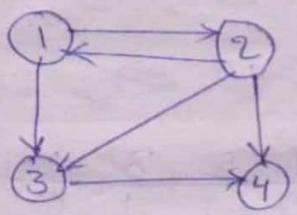
Scanned with CamScanner

- If the given node is not present in given graph, then return node not available.

(iv) Insert edge:-

```
addEdge(Vs, Ve);
```

where., Vs → Starting vertex
Ve → Ending vertex.



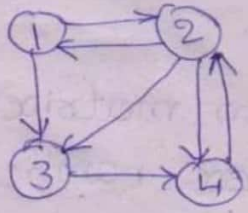
	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	0	0	0	1
4	0	0	0	0

This function used to insert an edge b/w two vertices. Those are.,

Vs → Starting vertex of edge.
Ve → Ending vertex of edge.

If two vertices that specified in addEdge are available in given graph G. then we put the value

```
G[Vs][Ve] = cost of edge
```



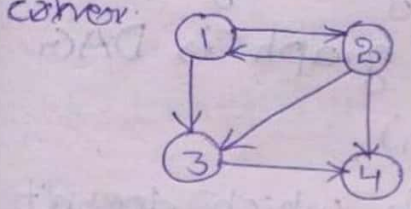
G[4][2] = 1

	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	0	0	0	1
4	0	1	0	0

Scanned with CamScanner

Delete edge:-

```
deleteEdge (Vs, Ve);
```



	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	0	0	0	1
4	0	0	0	0

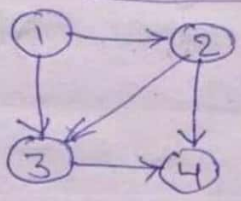
This function used to delete an edge b/w two vertices. Those are.,

Vs → starting vertex

Ve → Ending vertex

If two vertices that specified in deleteEdge are available in given graph G then we put the value.

```
G[Vs][Ve] = cost of edge
```



```
G[2][1] = 0;
```

	1	2	3	4
1	0	1	1	0
2	0	0	1	1
3	0	0	0	1
4	0	0	0	0

Graph Traversing:-

Graph traversing is the process of visiting every node (vertex) exactly once.

- Graph traversing is for
  - Finding a particular node available or not in a given graph.

Scanned with CamScanner

b) Finding all reachable nodes.
c) Finding best reachable nodes.
d) Finding best path through a graph.
e) Determining whether a graph is DAG. (directed acyclic graph)

DAG:- A directed graph which doesn't contains a cycles.

- The main goal of graph traverse is to find all nodes reachable from a given node
* In undirected graph we follow all edges
* In directed graph we follow outedge
- Graph traversing has two strategies/algorithms
1) DFS (Depth First search algorithm).
2) BFS (Breadth " " ").

DFS :—

Introduction :—

Depth First search / Traversing

DFS / Traversing was investigated in 19th century. by a french mathematician, charles Tremaux for solving maze.

Use of DFS

Definition :—

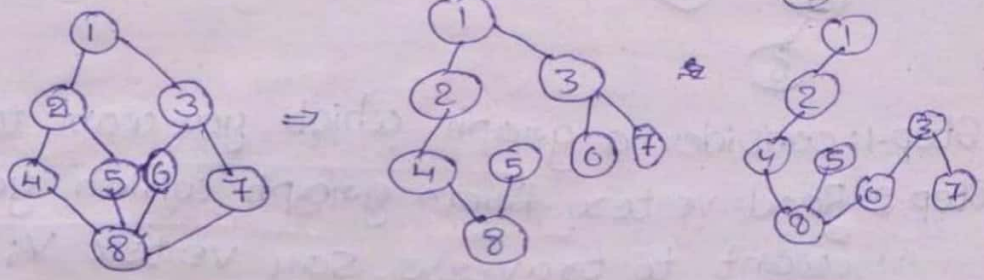
DFS is an algorithm for traversing / searching / tree data structure.

Working of DFS :—

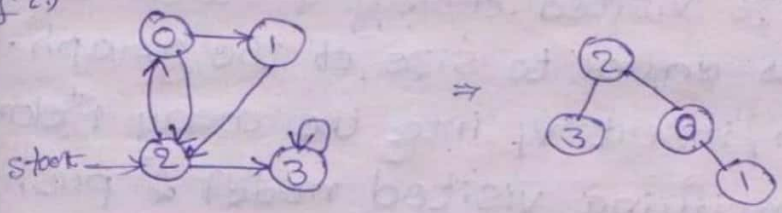
Scanned with CamScanner

In these start at the root (selecting some arbitary node as the root in the graph) & explore's (moves / visit). As far as possible along each branch before back-tracking.

Eg:1)



Eg-2)



- Spanning tree:—

Sub-graph N of a graph G is called Spanning tree if it satisfy the following two properties.

i) It includes all vertices/nodes of a graph G.

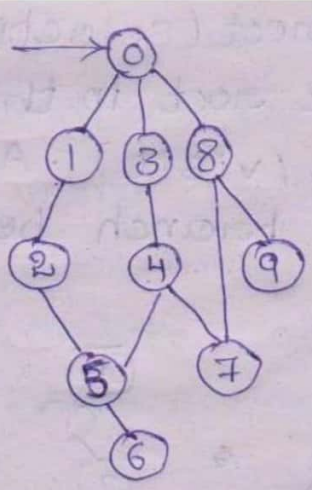
ii) There is no cycle's (It must be a tree).

We start from one vertex & traverse the path as deeply as we go. When there is no vertex for that then we traverse back & search for unvisited vertex.

Implementation of DFS:—

DFS is implemented by using a stack data structure.

Scanned with CamScanner



Step-1: consider a graph which you want to traverse

Step-2: Read vertex from graph which you want to traverse, say vertex V_i.

Step-3: Initialize visited array & stack. Array size is equal to size of the graph.

Step-4: Assign / insert V_i into the array 1st element (for specifying visited node) & push all adjacent nodes / vertices of V_i into the stack.

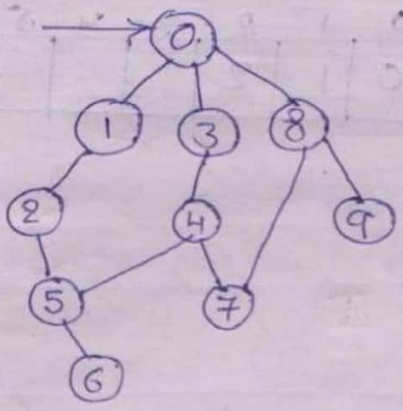
Step-5: Pop the top value of the stack & insert it to the visited array & push all adjacent nodes of poped node / element.

Step-6: If pop value (top of stack) is already present in array then don't insert into visited array, just we discard (or) neglect the top value.

Step-7: Do step-5 until stack is empty & array is full.

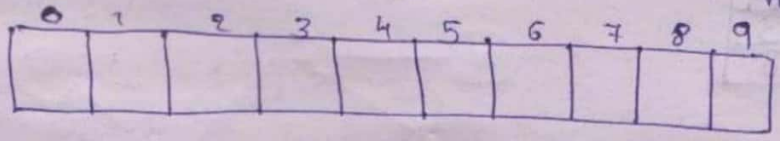
Scanned with CamScanner

Step-1:-

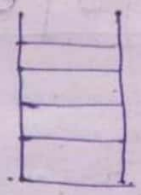


Step-2:- Here $v_i = 0$

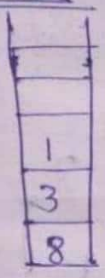
Step:3: visited array (Length of visited array is no. of nodes in the graph).

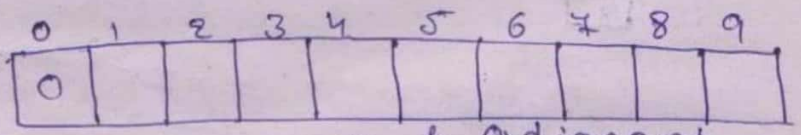


Take stack. Initially stack is empty



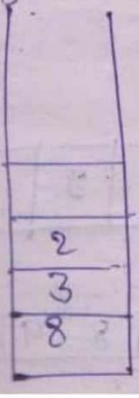
Step-4:-

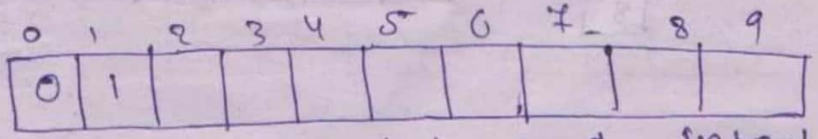




Add 0 to array & adjacent vertices of 0 to stack (1, 3, 8)

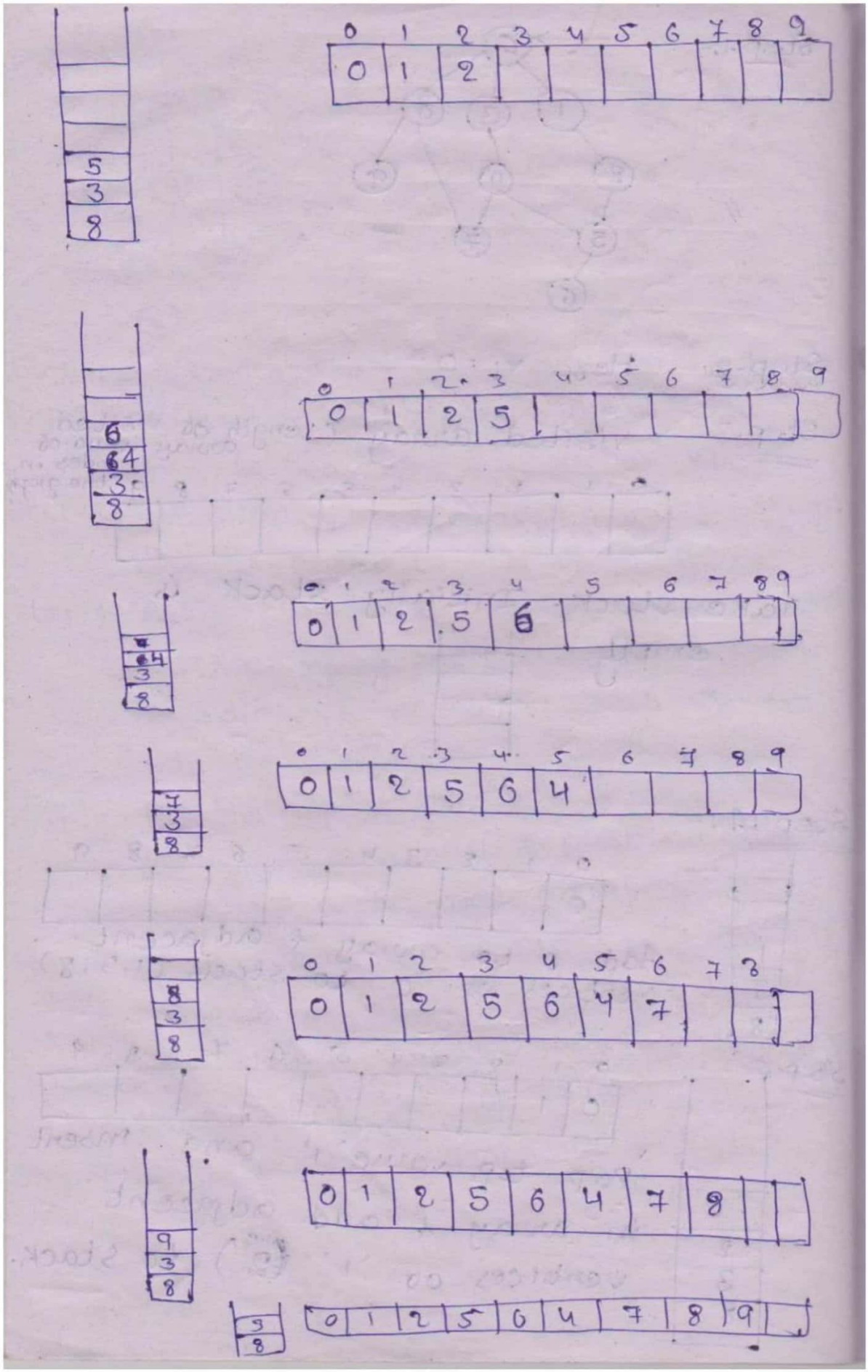
Step-5:



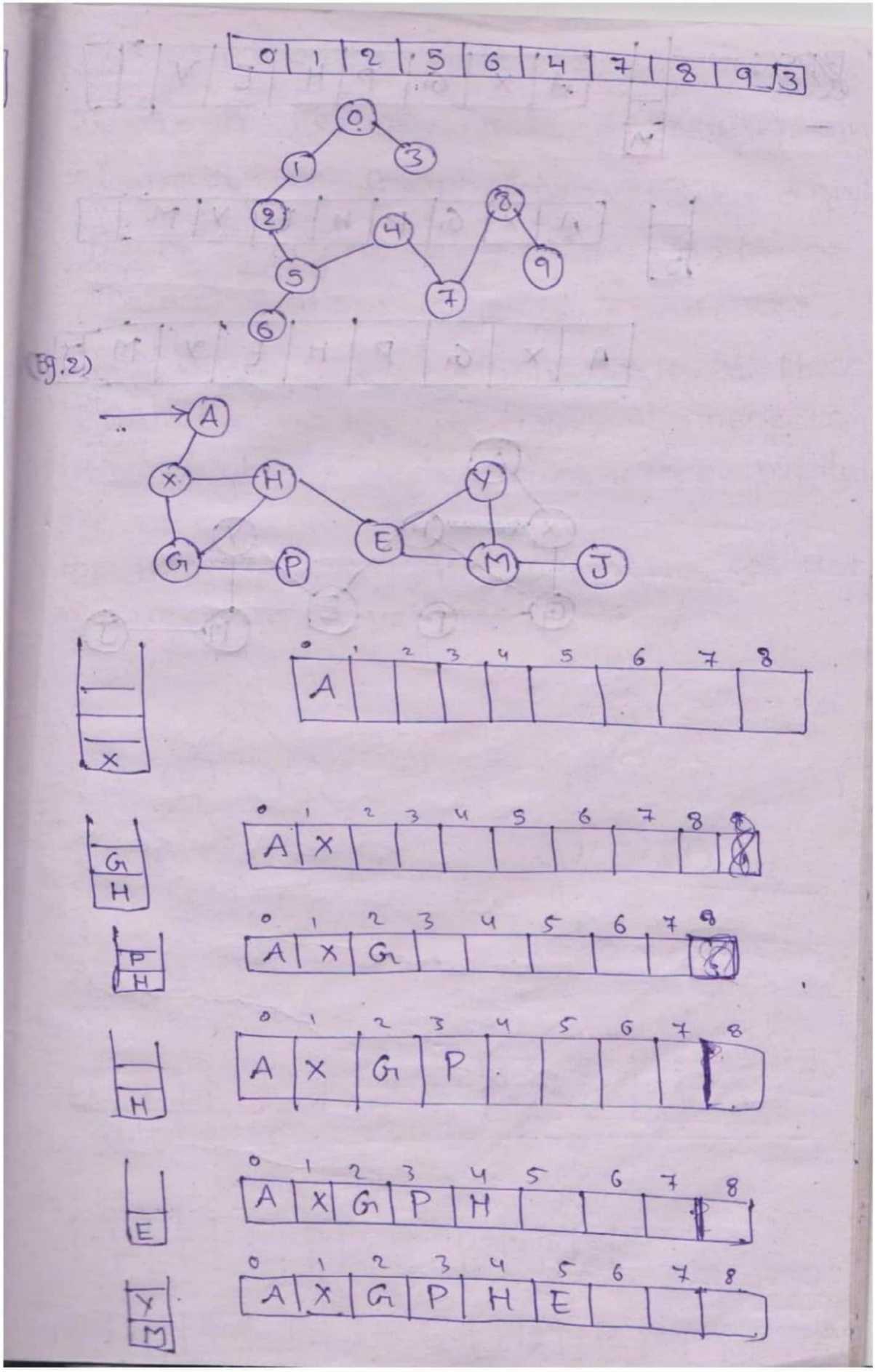


Pop top value '1' and insert in array & add adjacent vertices of '1' (i.e. 2) to stack.

Scanned with CamScanner

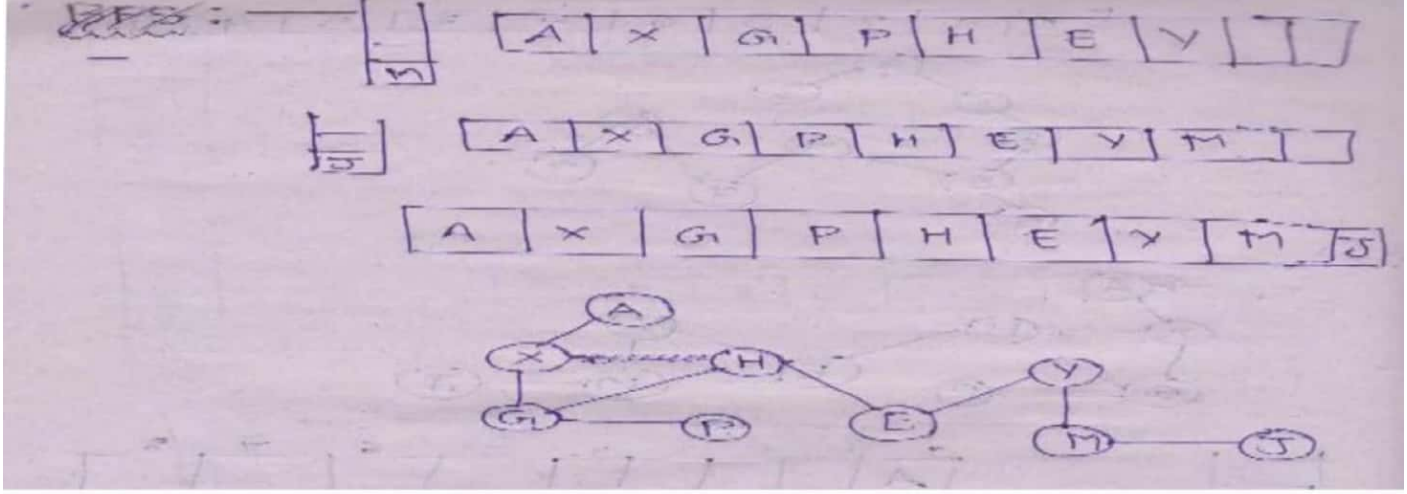


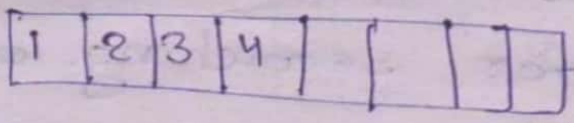
Scanned with CamScanner

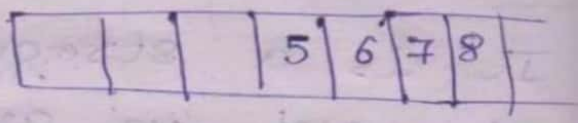


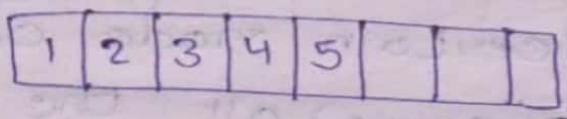
Scanned with CamScanner

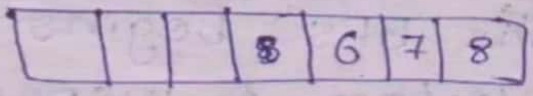
BFS :

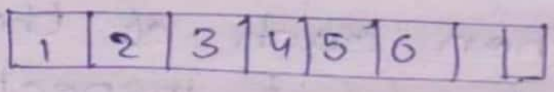


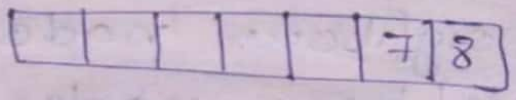


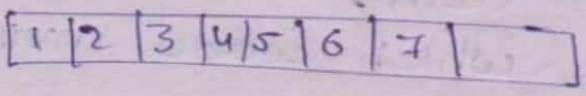


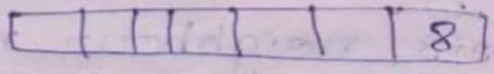


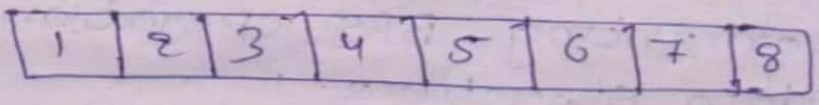












Algorithm for BFS:—

Step-1: consider the graph which you want to find the vertex (Graph traversing)

Step-2: select any vertex called V_i in our graph where ~~you~~ we want to start graph traversing

Step-3: consider any two data structures,
- visited array (size of the graph.
- queue d.s (FIFO)

Step-4: Assign starting vertex V_i into the visited array & the adjacent vertices or adjacent nodes of V_i are insert into the queue.

Step-5: Now pop element of queue by using FIFO principle, that popped element

---

insert into the array second element
The popped element adjacent vertices
inserted into queue.

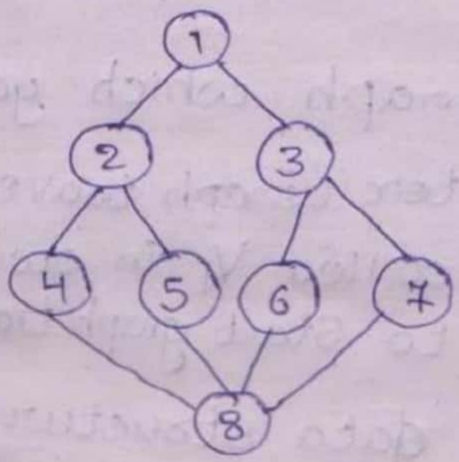
Step-6: Do Step-~~4~~5 until there is no vertex left in graph & there is no loop.

BFS:— (Breadth first search)

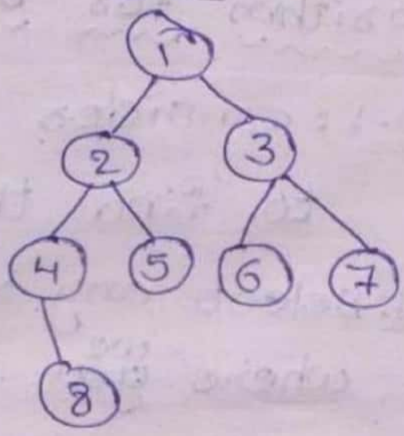
It is a stragey for searching an element in graph (traversing a graph)

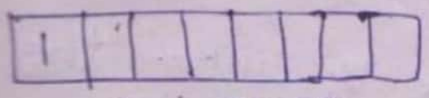
- The BFS begins at certain node or vertex. & inspects (observe) all the neighbour nodes. Then each of those neighbour nodes in turn, it inspects their neighbour nodes which were invisited and so on....
- For implementing BFS operation we use queue data structure.

Ex:-

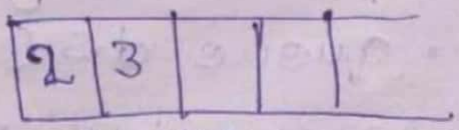


BFS

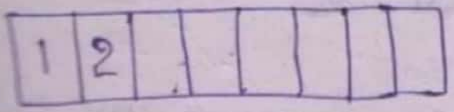


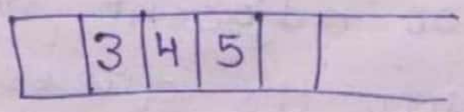


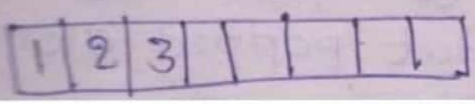
visited array

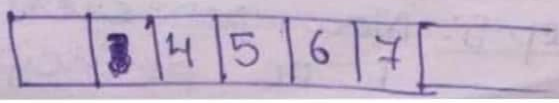


queue.









Connected Component Definition

A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.

Let's try to simplify it further, though. A set of nodes forms a connected component in an undirected graph if any node from the set of nodes can reach any other node by traversing edges. **The main point here is reachability.**

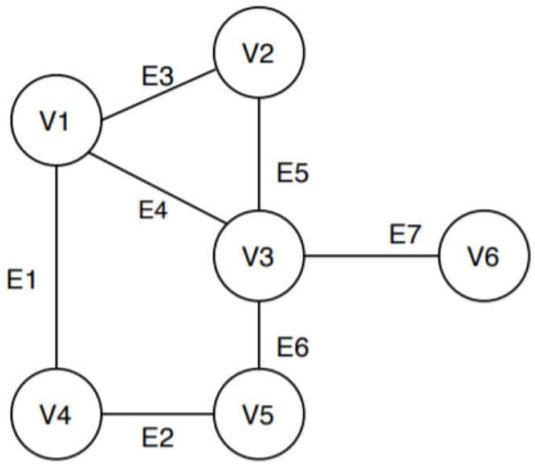
In connected components, all the nodes are always reachable from each other.

Examples

In this section, we'll discuss a couple of simple examples. We'll try to relate the examples with the definition given above.

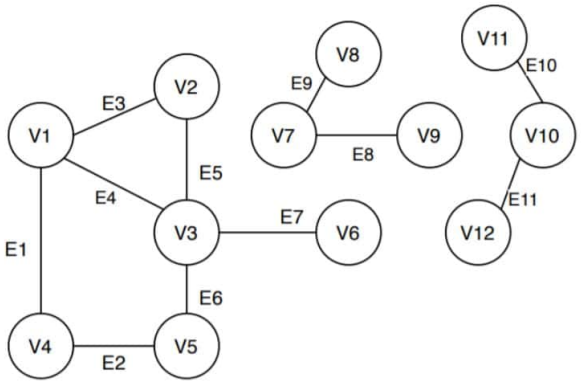
One Connected Component

In this example, the given undirected graph has one connected component:



More Than One Connected Component

In this example, the undirected graph has three connected components:



The graph has 3 connected components

BI-CONNECTED components

A vertex v of G is called an articulation point, if removing v along with the edges incident on v, results in a graph that has at least two connected components.

A bi-connected graph (shown in Fig. 13.10) is defined as a connected graph that has no articulation vertices. That is, a bi-connected graph is connected and non-separable in the sense that even if we remove any vertex from the graph, the resultant graph is still connected. By definition,

1. A bi-connected undirected graph is a connected graph that **cannot be broken into disconnected pieces by deleting any single vertex**.
2. In a bi-connected directed graph, for any two vertices v and w, there are two directed paths from v to w which have no vertices in common other than v and w. Note that the graph shown in Fig. 13.9(a) is not a bi-connected graph, as deleting vertex C from the graph results in two disconnected components of the original graph (Fig. 13.9(b)).

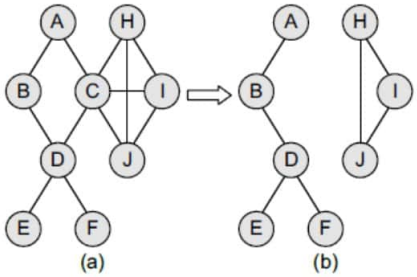


Figure 13.9 Non bi-connected graph

vertices, there is a related concept for edges. An edge in a graph is called a *bridge* if removing that edge results in a disconnected graph. Also, an edge in a graph that does not lie on a cycle is a bridge. This means that a bridge has at least one articulation point at its end, although it is not necessary that the articulation point is linked to a bridge.

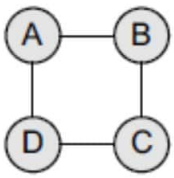


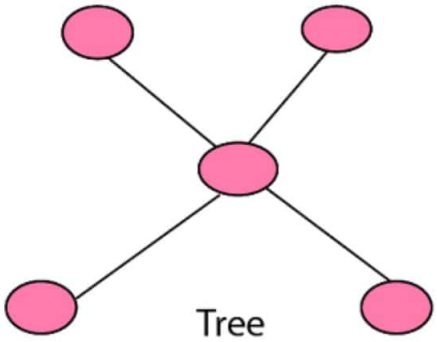
Figure 13.10 Bi-connected graph

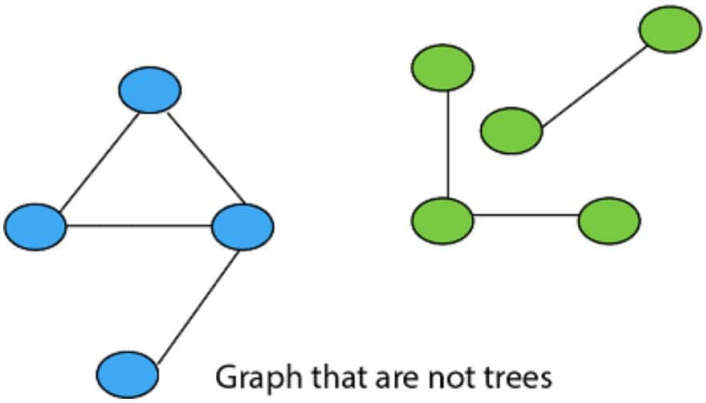
Minimum Spanning Tree:

Tree:

A tree is a graph with the following properties:

1. The graph is connected (can go from anywhere to anywhere)
2. There are no cyclic (Acyclic)



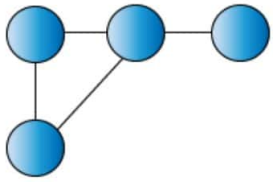


Spanning Tree:

Given a connected undirected graph, a spanning tree of that graph is a sub graph that is a tree and joined all vertices. A single graph can have many spanning trees.

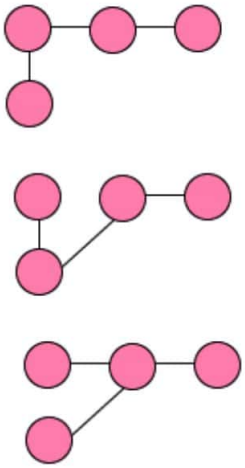
For Example:

Connected Undirected Graph



For the above-connected graph. There can be multiple spanning Trees like

Spanning Trees



Properties of Spanning Tree:

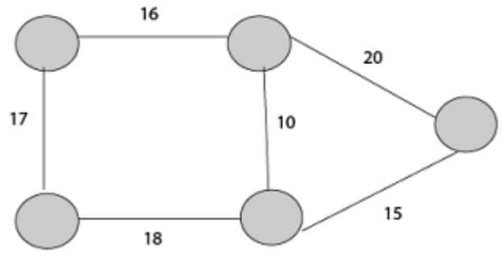
1. There may be several minimum spanning trees of the same weight having the minimum number of edges.
2. If all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
3. If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.
4. A connected graph G can have more than one spanning trees.
5. A disconnected graph can't have to span the tree, or it can't span all the vertices.
6. Spanning Tree doesn't contain cycles.

7. Spanning Tree has **(n-1) edges** where n is the number of vertices.

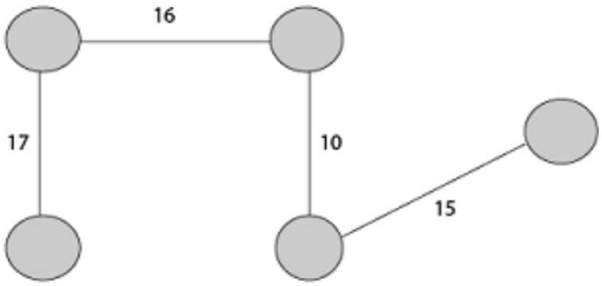
Addition of even one single edge results in the spanning tree losing its property of **Acyclicity** and elimination of one single edge results in its losing the property of connectivity.

Minimum Spanning Tree:

Minimum Spanning Tree is a Spanning Tree which has minimum total cost. If we have a linked undirected graph with a weight (or cost) combine with each edge. Then the cost of spanning tree would be the sum of the cost of its edges.



Connected , Undirected Graph



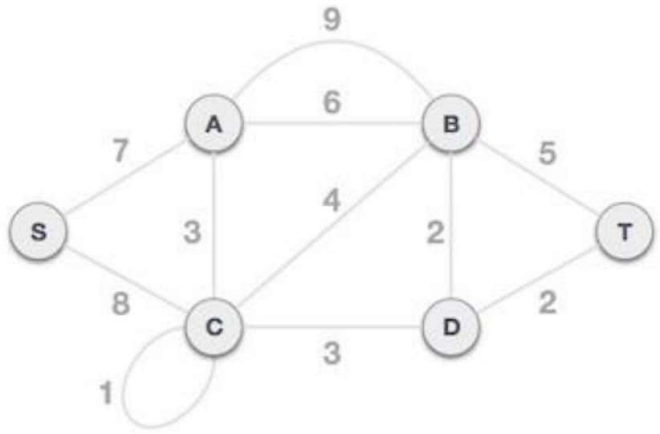
Minimum Cost Spanning Tree

Total Cost = 17+16+10+15=58

Kruskal's Algorithm

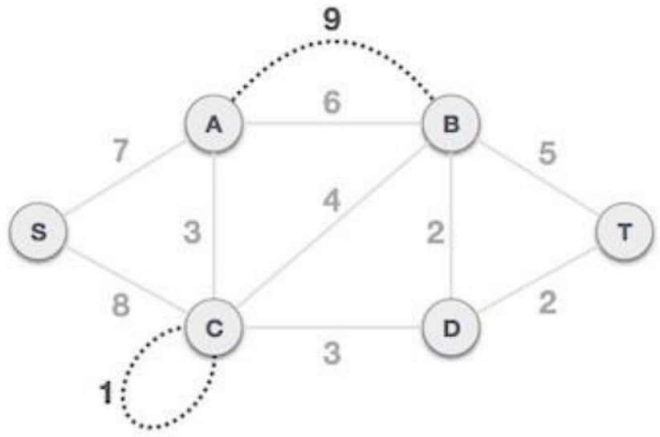
Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph. Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

Example

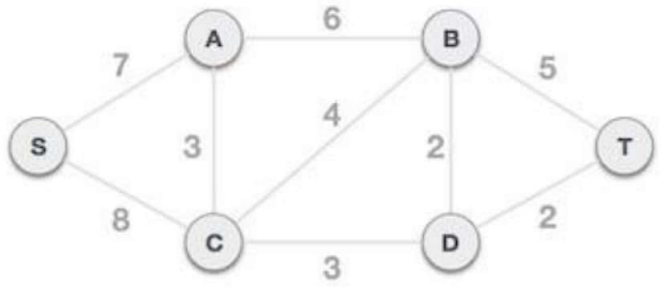


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



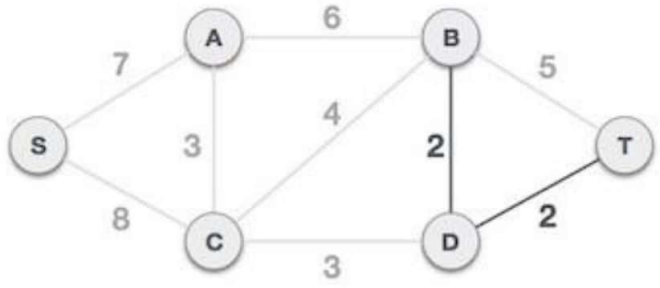
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

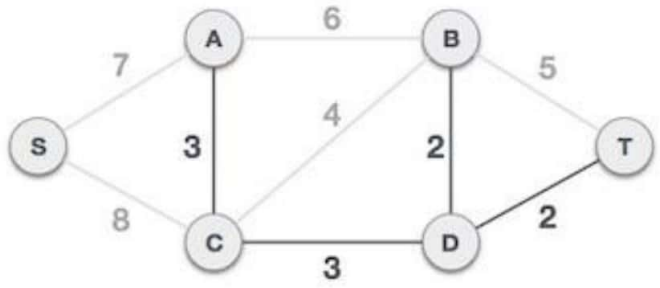
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

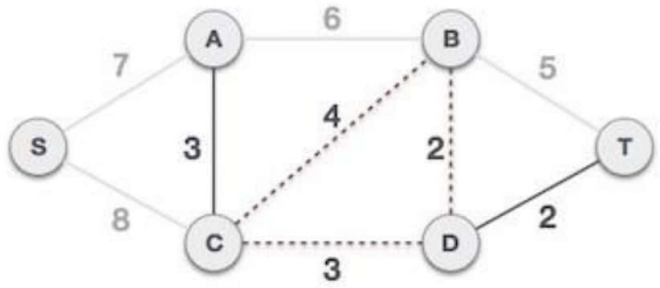


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

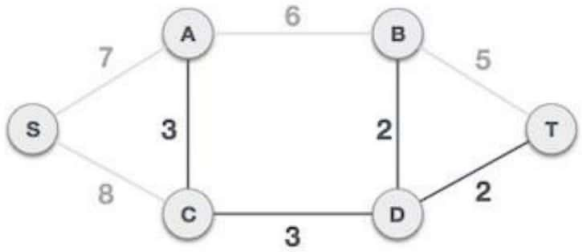
Next cost is 3, and associated edges are A,C and C,D. We add them again −



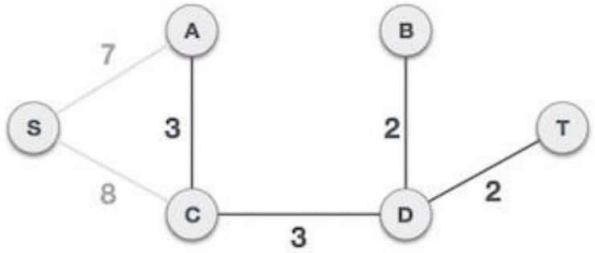
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



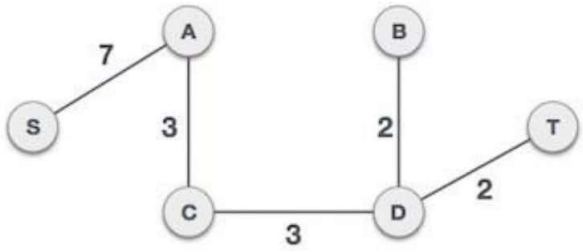
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



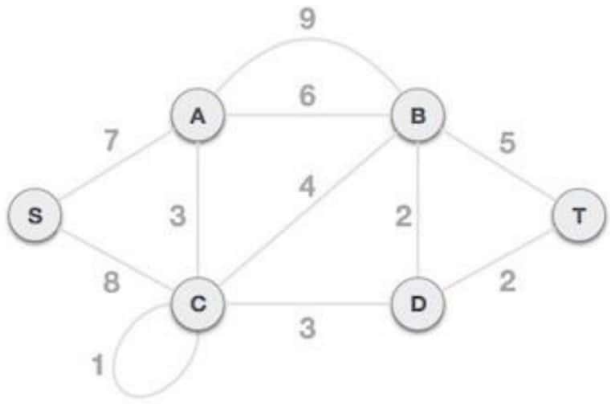
By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Prim's algorithm:

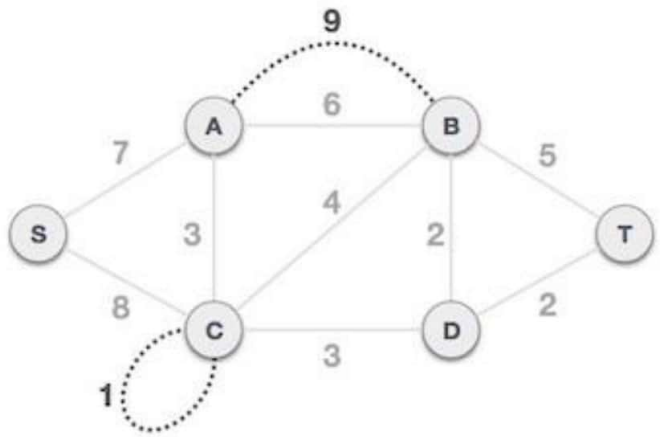
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

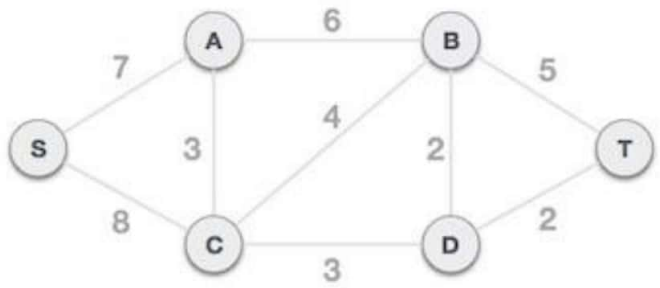
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

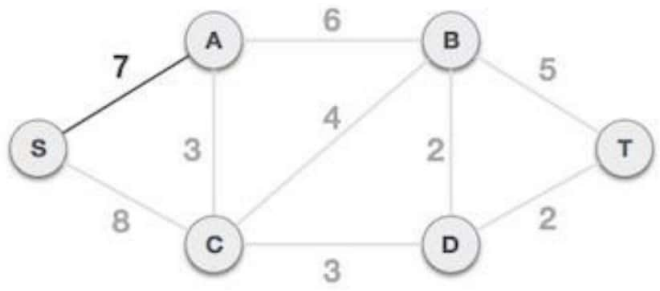


Step 2 - Choose any arbitrary node as root node

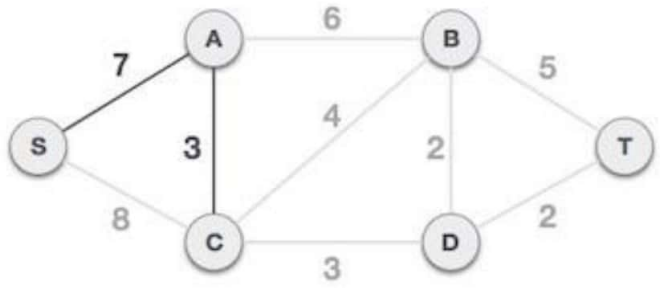
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

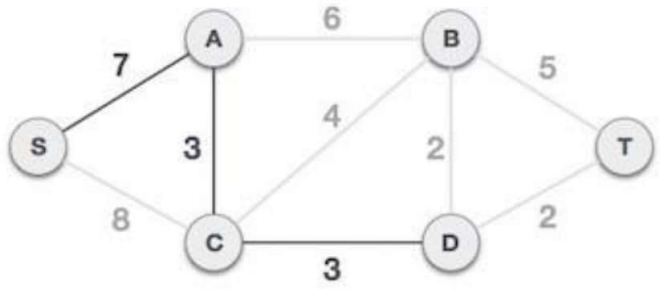
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



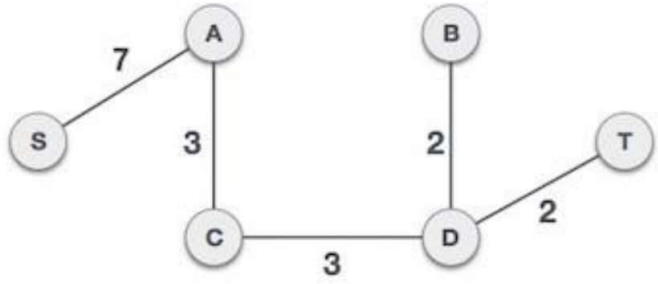
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.

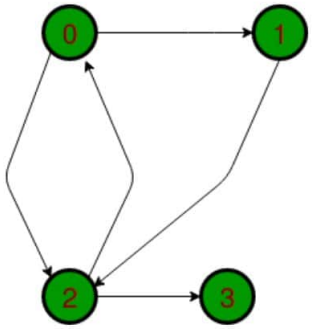


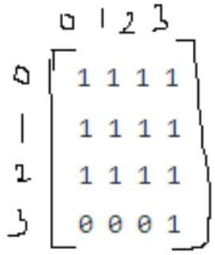
We may find that the output spanning tree of the same graph using two different algorithms is same.

Transitive closure of a graph

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j. The reach-ability matrix is called the transitive closure of a graph.

For example, consider below graph





Transitive closure of above graphs is

Shortest path:

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

(OR)

The problem of finding the **shortest path** in a graph from one vertex to another vertex. "**Shortest**" may be least number of edges, least total weight, etc. Also known as single-pair **shortest-path** problem.

All Pair Shortest Path Algorithm(Warshall algorithm):

The all pair shortest path algorithm is also known as Floyd- Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

Algorithm:

Step-01:

- Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self edges nor parallel edges.

Step-02:

- Write the initial distance matrix.
- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞.

Initial distance matrix for the given graph is-

$$D_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

Step-03:

Using Floyd Warshall Algorithm, write the following 4 matrices-

Now, create a matrix D1 using matrix D0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

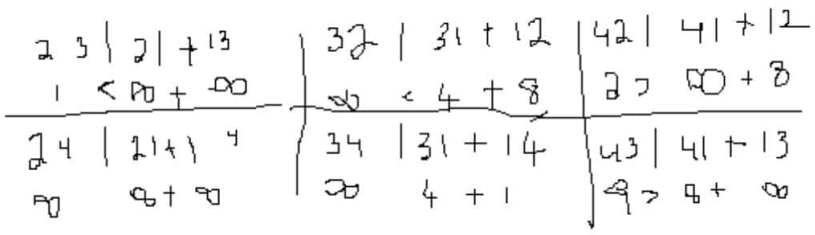
Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. D[i][j] is filled if (D[i][j] < D[i][k] + D[k][j]).

That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with D[i][k] + D[k][j].

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

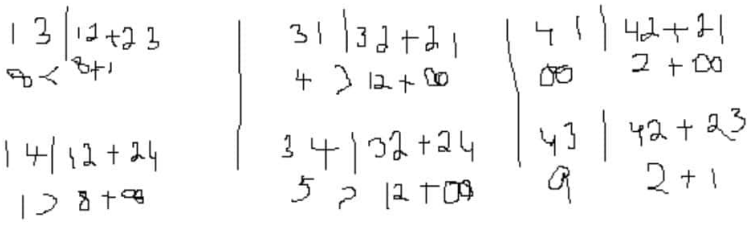
$D_1 =$

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	9	0



$D_2 =$

	1	2	3	4
1	0	8	9	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	3	0



$D_3 =$

	1	2	3	4
1	0	8	9	1
2	5	0	1	6
3	4	12	0	5
4	7	2	3	0

$D_4 =$

	1	2	3	4
1	0	3	4	1
2	5	0	1	6
3	4	7	0	5
4	7	2	3	0

The last matrix D_4 represents the shortest path distance between every pair of vertices.