GRAPHS

- **A graph is a non-linear data structure which is a collection of *vertices (also* called *nodes) and edges that connect these vertices.***

USE:

Graphs are widely used to model any situation where entities or things are related to each other in pairs.

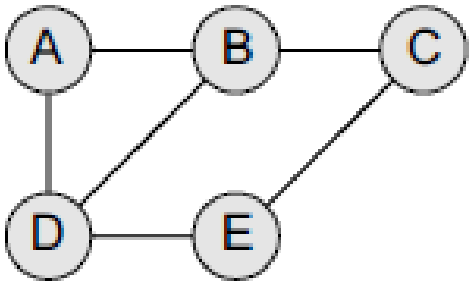
For example:

- *Family trees in which the member nodes have an edge from parent to each of their children.*
- *Transportation networks in which nodes are airports, intersections, ports, etc. The edges can* be airline flights, one-way roads, shipping routes, etc.

GRAPHS

Definition :

A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.



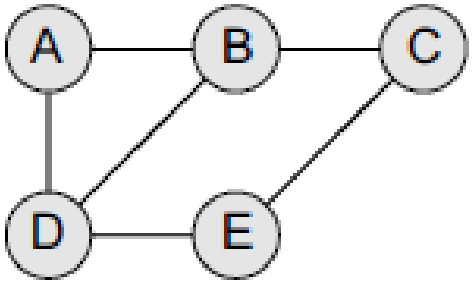
Undirected graph

Above Figure shows a graph with V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}. Note that there are five vertices or nodes and six edges in the graph.

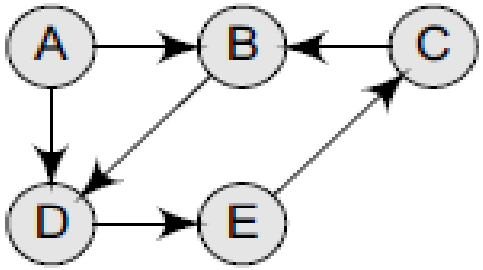
GRAPHS

→ A graph can be directed or undirected.

In an ***undirected graph***, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. Figure 1 shows an undirected graph because it does not give any information about the direction of the edges.



1. Undirected graph



2. Directed graph

Fig. 2 which shows a ***directed graph***. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

GRAPHS

Graph Terminology

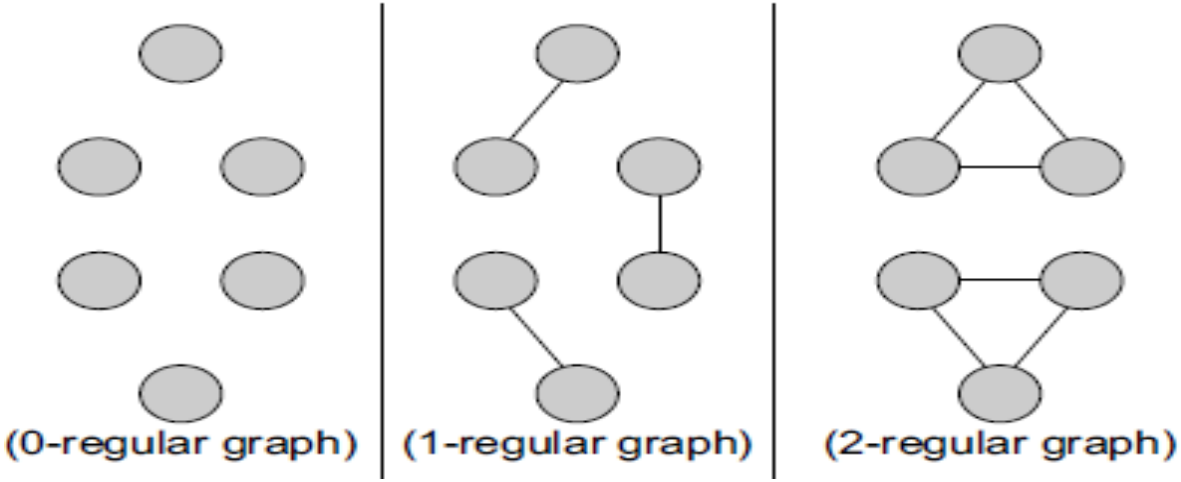
Adjacent nodes or neighbours: *For every edge, e = (u, v) that connects nodes u and v, the nodes* u and v are the end-points and are said to be the adjacent nodes or neighbours.

Degree of a node: *Degree of a node u, deg(u), is the total number of edges containing the node u.* If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an **isolated node**.

GRAPHS

Graph Terminology

Regular graph: *It is a graph where each vertex has the* same number of neighbours. That is, every node has the same degree. A regular graph with vertices of degree k is called a **k–regular graph or a regular graph of degree k.**



Regular graphs

GRAPHS

Graph Terminology

Path: *A path P written as P = {v0, v1, v2, ..., vn), of length* n from a node u to v is defined as a sequence of (n+1) nodes.

Closed path: *A path P is known as a closed path if the edge has the same end-points. That is, if* v0 = vn.

Simple path: *A path P is known as a simple path if all the nodes in the path are distinct with an* exception that v0 may be equal to vn. If v0 = vn, then the path is called a closed simple path.

GRAPHS

Graph Terminology

Cycle: *A path in which the first and the last vertices are same. A simple cycle has no repeated* edges or vertices (except the first and last vertices).

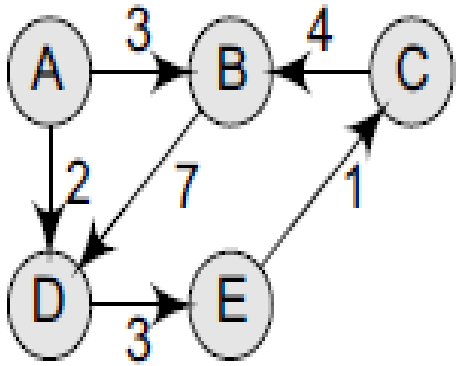
Connected graph: A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree.

GRAPHS

Graph Terminology

Complete graph: *A graph G is said to be complete if all its nodes are fully connected. That is,* there is a path from one node to every other node in the graph. A complete graph has n(n–1)/2 edges, where n is the number of nodes in G.

Labelled graph or weighted graph: *A graph is said to be labelled if every edge in the graph* is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge. Below Figure shows a weighted graph.



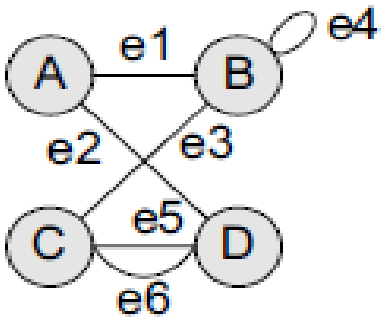
GRAPHS

Graph Terminology

Multiple edges: *Distinct edges which connect the same end-points are called multiple edges.* That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

Loop: An edge that has identical end-points is called a loop. That is, e = (u, u).

Multi-graph: A graph with multiple edges and/or loops is called a multi-graph.



(a) Multi-graph

Size of a graph: The size of a graph is the total number of edges in it.

GRAPHS

Directed Graphs

A directed graph G, also known as a *digraph, is a graph in which every edge has a direction assigned* to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G.
For an edge (u, v)

- The edge begins at u and terminates at v.
- u is known as the origin or initial point of e. Correspondingly, v is known as the destination or terminal point of e.
- u is the predecessor of v. Correspondingly, v is the successor of u.
- Nodes u and v are adjacent to each other.

GRAPHS

Terminology of a Directed Graph

Out-degree of a node: *The out-degree of a node u, written as outdeg(u), is the number of edges* that originate at u.

In-degree of a node: *The in-degree of a node u, written as indeg(u), is the number of edges that* terminate at u.

Degree of a node: *The degree of a node, written as deg(u), is equal to the sum of in-degree and* out-degree of that node. Therefore, deg(u) = indeg(u) + outdeg(u).

Isolated vertex: *A vertex with degree zero. Such a vertex is not an end-point of any edge.*

GRAPHS

Terminology of a Directed Graph

Pendant vertex (also known as leaf vertex): *A vertex with degree one.*

Cut vertex: *A vertex which when deleted would disconnect the remaining graph.*

Source: *A node u is known as a source if it has a positive out-degree but a zero in-degree.*

Sink: *A node u is known as a sink if it has a positive in-degree but a zero out-degree.*

GRAPHS

Terminology of a Directed Graph

Reachability: *A node v is said to be reachable from node u, if and only if there exists a (directed)* path from node u to node v.

Strongly connected directed graph: *A digraph is said to be strongly connected if and only if there* exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.

Unilaterally connected graph: *A digraph is said to be unilaterally connected if there exists a path* between any pair of nodes u, v in G such that there is a path from u to v or a path from v to u, but not both.

GRAPHS

Terminology of a Directed Graph

Weakly connected digraph: *A directed graph is said to be weakly connected if it is connected* by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

Parallel/Multiple edges: Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

GRAPHS

REPRESENTATION OF GRAPHS

There are three common ways of storing graphs in the computer's memory. They are:

1. *Sequential representation by using an adjacency* matrix.

2. *Linked representation by using an adjacency list that stores the neighbours of a node using* a linked list.

3. *Adjacency multi-list which is an extension of linked representation.*

GRAPHS

REPRESENTATION OF GRAPHS

Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them.

In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v.

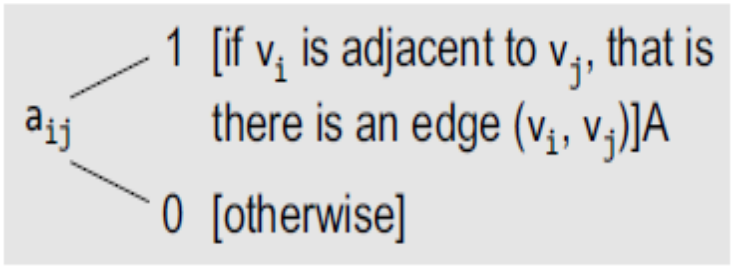
For any graph G having n nodes, the adjacency matrix will have the dimension of n X n.

In an adjacency matrix, the rows and columns are labelled by graph vertices.

GRAPHS

REPRESENTATION OF GRAPHS

Adjacency Matrix Representation



Adjacency matrix entry

An entry a_{ij} in the adjacency matrix will contain 1, if vertices v_i and v_j are adjacent to each other.

However, if the nodes are not adjacent, a_{ij} will be set to zero.

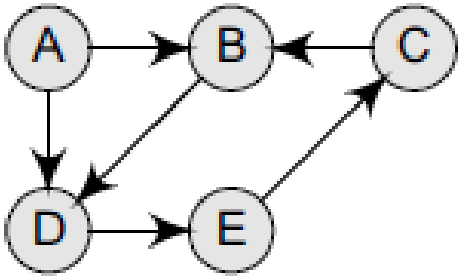
Since an adjacency matrix contains only 0s and 1s, it is called a *bit matrix or a Boolean matrix.*

The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.

GRAPHS

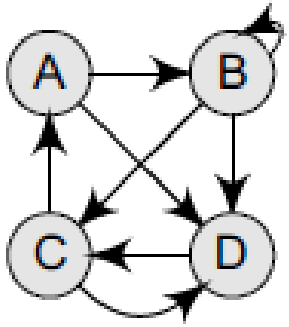
REPRESENTATION OF GRAPHS

Adjacency Matrix Representation



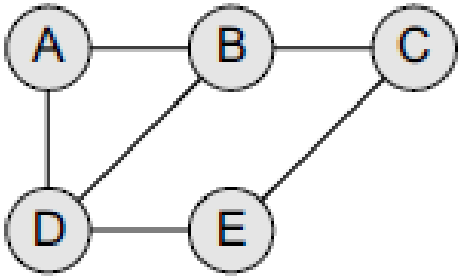
	A	B	C	D	E
A	0	1	0	1	0
B	0	0	0	1	0
C	0	1	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

(a) Directed graph



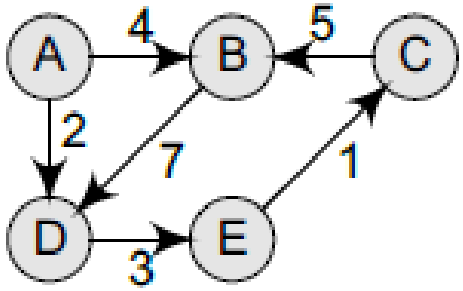
	A	B	C	D
A	0	1	0	1
B	0	1	1	1
C	1	0	0	1
D	0	0	1	0

(b) Directed graph with loop



	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

(c) Undirected graph



	A	B	C	D	E
A	0	4	0	2	0
B	0	0	0	7	0
C	0	5	0	0	0
D	0	0	0	0	3
E	0	0	1	0	0

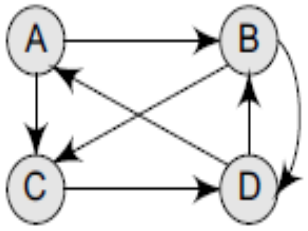
(d) Weighted graph

GRAPHS

REPRESENTATION OF GRAPHS

Adjacency Matrix (powers of an adjacency matrix)

$$A^2 = A^1 \times A^1$$



$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

$$A^2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

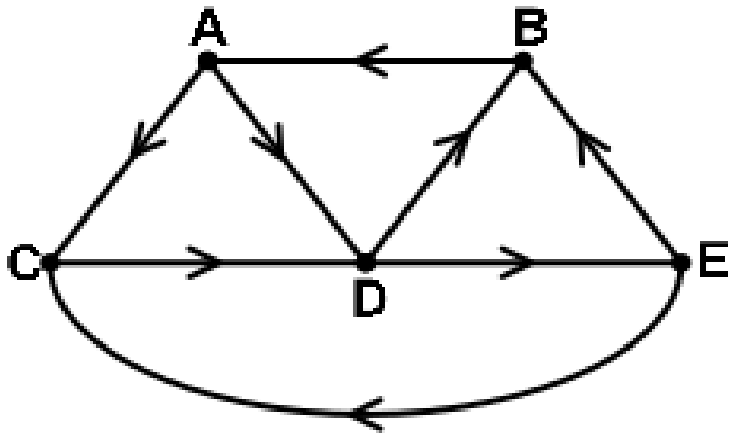
$$A^3 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

GRAPHS

REPRESENTATION OF GRAPHS

GTU Question:

Obtain the adjacency matrix A for the following graph.
Find A^2.
Find outdegree of E and D nodes.

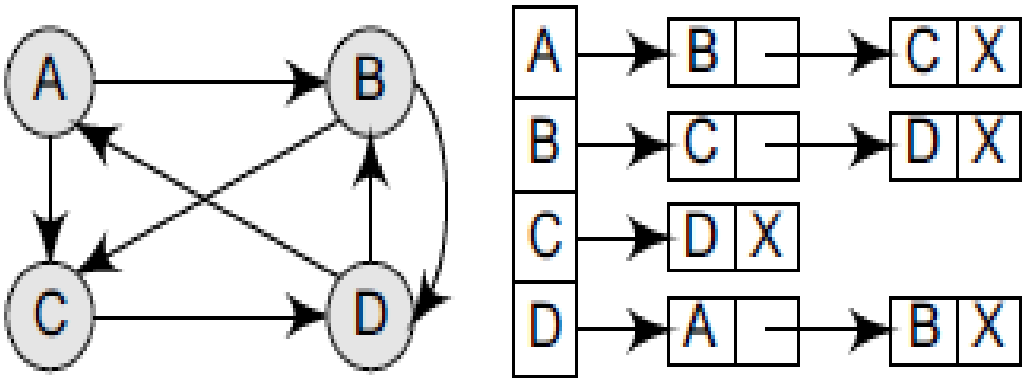


GRAPHS

REPRESENTATION OF GRAPHS

Adjacency List Representation

An adjacency list is another way in which graphs can be represented in the computer's memory.
This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.



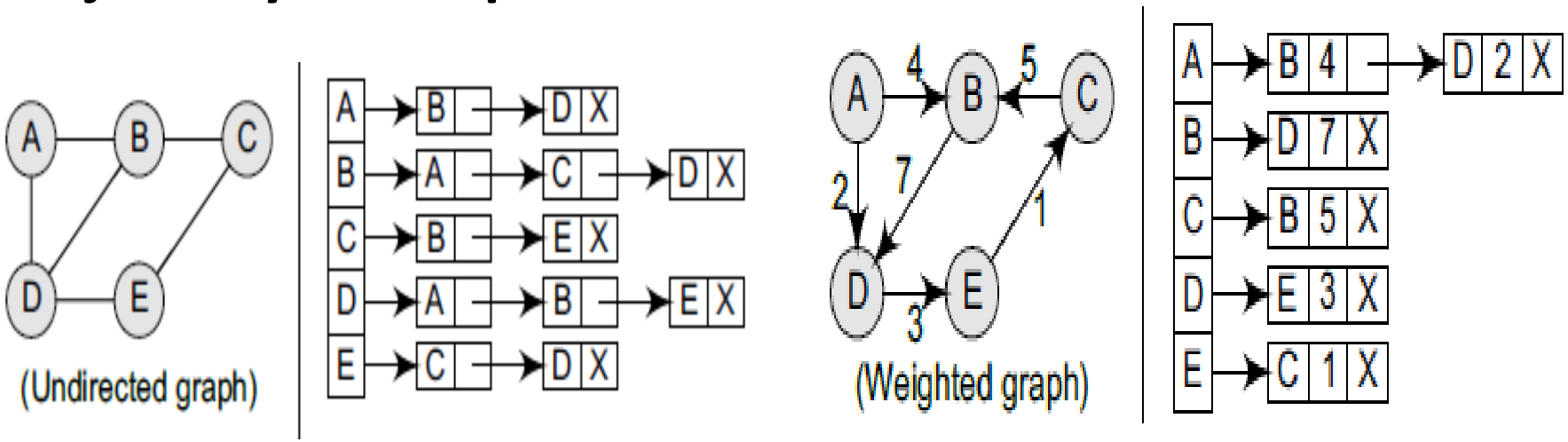
Graph G and its adjacency list

For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G.

GRAPHS

REPRESENTATION OF GRAPHS

Adjacency List Representation



Adjacency list for an undirected graph and a weighted graph

However, for an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge from v to u. Adjacency lists can also be modified to store weighted graphs.

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

There are two standard methods of graph traversal

1. **Breadth-first search (BFS)**
2. **Depth-first search (DFS)**

While **breadth-first search uses a queue** as an auxiliary data structure to store nodes for further processing, the **depth-first search scheme uses a stack**. But both these algorithms make use of a variable **STATUS**.

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

During the execution of the algorithm, every node in the graph will have the variable STATUS set to 1, 2 or 3, depending on its current state.

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

Value of status and its significance

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Breadth-First Search Algorithm

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes.

Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.

That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth.

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Breadth-First Search Algorithm

This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

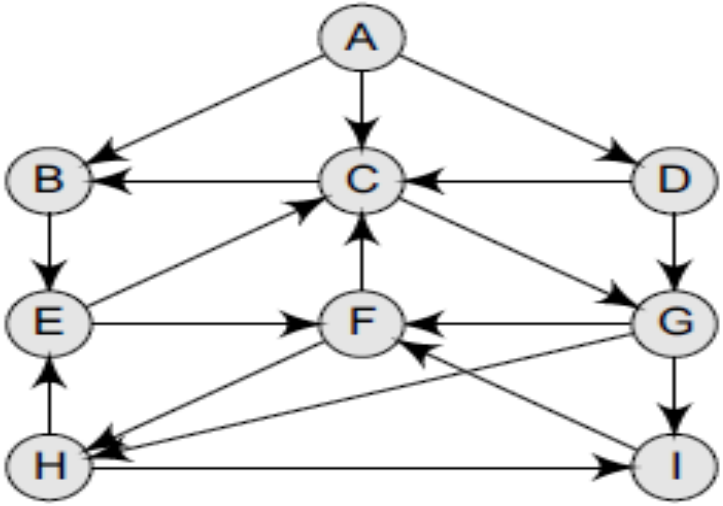
GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Breadth-First Search Algorithm

Example:

Consider the graph G given in below Fig. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.



Adjacency lists

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

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Solution:

•The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays: QUEUE and ORIG.

•While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge. Initially, FRONT = REAR = –1. The algorithm for this is as follows:

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

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

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

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

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

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

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

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

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

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

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

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

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

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

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

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

FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 8	ORIG = \0	A	A	A	B	C	E	G	G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE.

Now, backtrack from I using ORIG to find the minimum path P. Thus, we have P as A -> C -> G -> I.

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Applications of Breadth-First Search Algorithm

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

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

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Depth-first Search Algorithm

•The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

•In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Depth-first Search Algorithm

The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the deadend, we backtrack to find another path P. The algorithm terminates when backtracking leads back to the starting node A. In this algorithm, edges that lead to a new vertex are called *discovery edges and edges that* lead to an already visited vertex are called *back edges.*

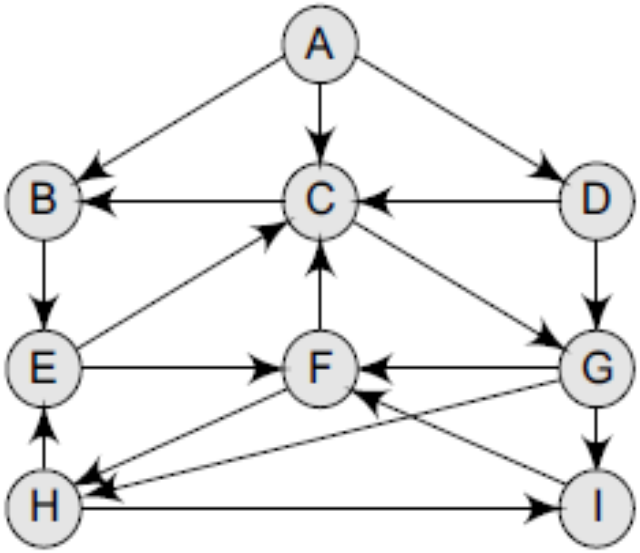
Observe that this algorithm is similar to the in-order traversal of a binary tree. Its implementation is similar to that of the breadth-first search algorithm but here we use a stack instead of a queue. Again, we use a variable STATUS to represent the current state of the node.

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Example:

Consider the graph G given in Fig. The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H. The procedure can be explained here.



Adjacency lists

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

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Solution:

(a) Push H onto the stack.

STACK: H

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

PRINT: H

STACK: E, I

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

PRINT: I

STACK: E, F

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Solution:

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

PRINT: F | STACK: E, C

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

PRINT: C | STACK: E, B, G

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

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Solution:

PRINT: G | STACK: E, B

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

PRINT: B | STACK: E

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

PRINT: E | STACK:

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Solution:

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

H, I, F, C, G, B, E

These are the nodes which are reachable from the node H.

GRAPHS

GRAPH TRAVERSAL ALGORITHMS

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

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

GRAPHS

SHORTEST PATH ALGORITHMS

Minimum spanning tree

A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together.

A graph G can have many different spanning trees.

We can assign *weights* to each edge, and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning tree.

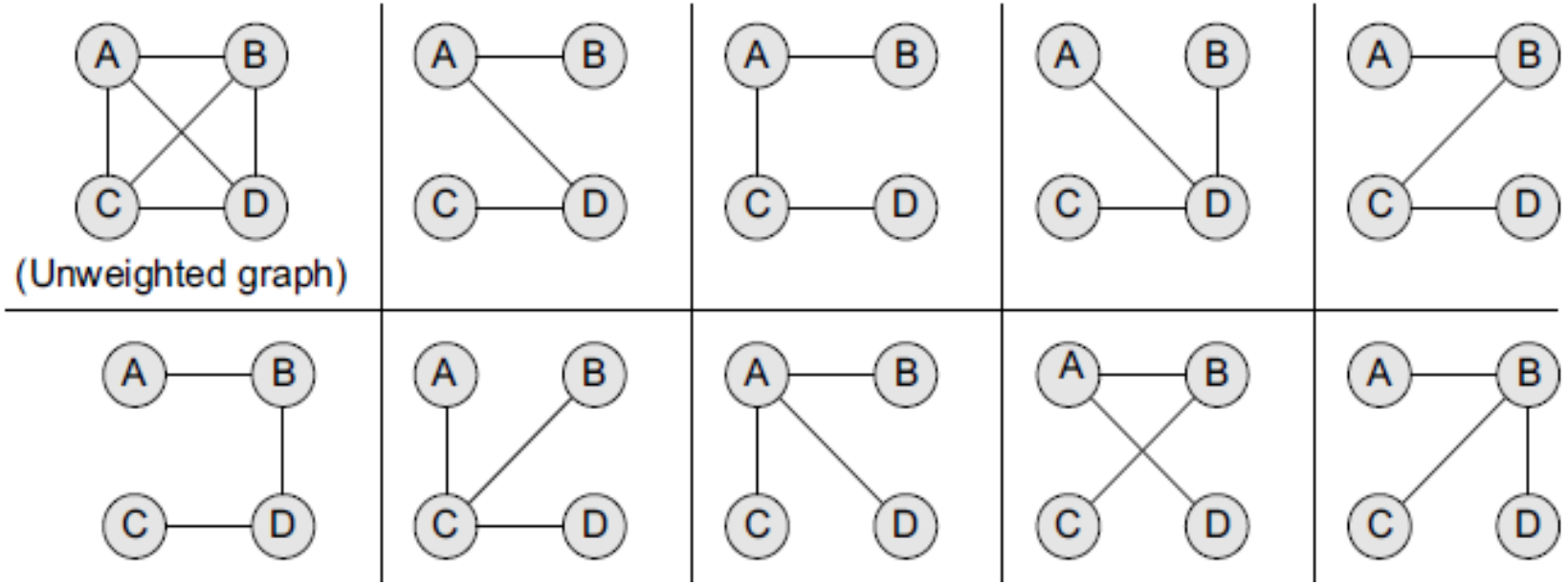
***A minimum spanning tree (MST) is defined as a spanning tree with weight less than or equal* to the weight of every other spanning tree. In other words, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.**

GRAPHS

SHORTEST PATH ALGORITHMS

Minimum spanning tree

Consider an unweighted graph G given below. From G, we can draw many distinct spanning trees. Eight of them are given here. For an unweighted graph, every spanning tree is a minimum spanning tree.

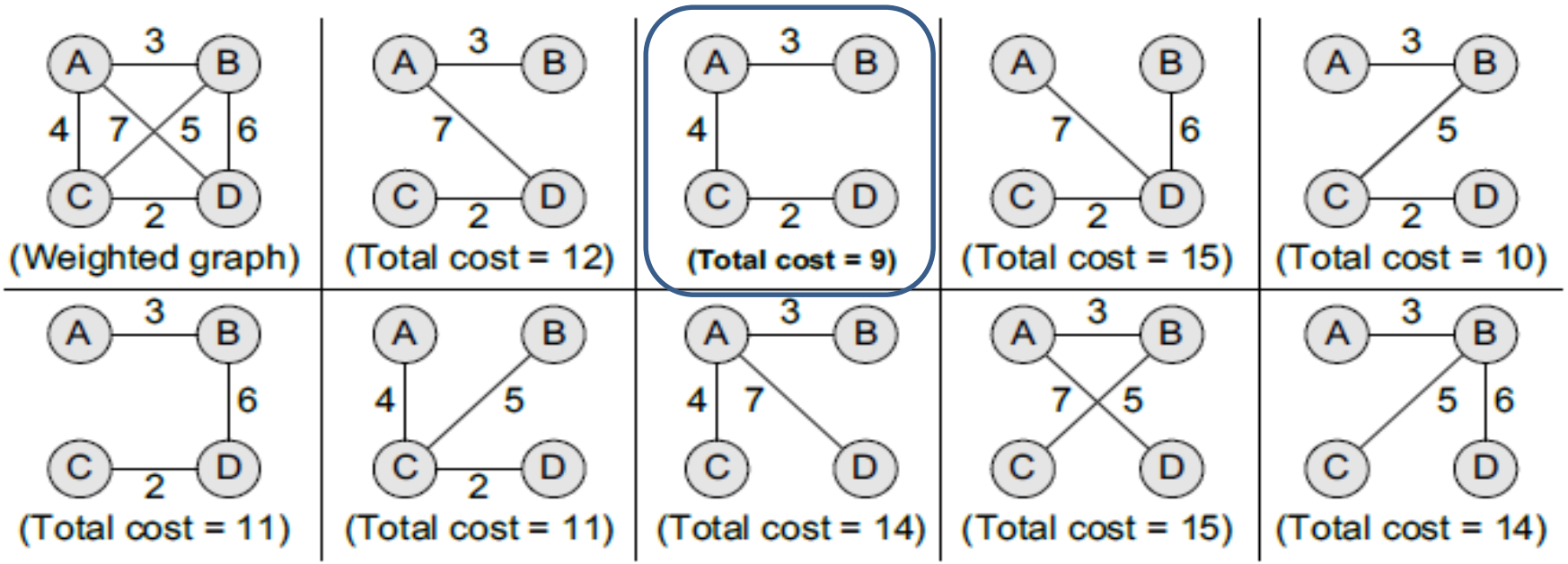


GRAPHS

SHORTEST PATH ALGORITHMS

Minimum spanning tree

Consider a weighted graph G given below. From G, we can draw many distinct spanning trees. But only a single minimum spanning tree can be obtained, that is, the one that has the minimum weight (cost) associated with it. One that is highlighted is called the minimum spanning tree, as it has the lowest cost (9) associated with it.



GRAPHS

Minimum spanning tree : Prim's Algorithm

Prim's algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph.
In other words, the algorithm builds a tree that includes every vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is minimized.

For this, the algorithm maintains three sets of vertices which can be given as below:

- ***Tree vertices*** *Vertices that are a part of the minimum spanning tree T.*
- ***Fringe vertices*** *Vertices that are currently not a part of T, but are adjacent to some tree vertex.*
- ***Unseen vertices*** *Vertices that are neither tree vertices nor fringe vertices fall under this category.*

GRAPHS

Minimum spanning tree : Prim's Algorithm

The steps involved in the Prim's algorithm are shown below:

Step 1: Select a starting vertex

Step 2: Repeat Steps 3 and 4 until there are fringe vertices

Step 3: Select an edge e connecting the tree vertex and fringe vertex that has minimum weight

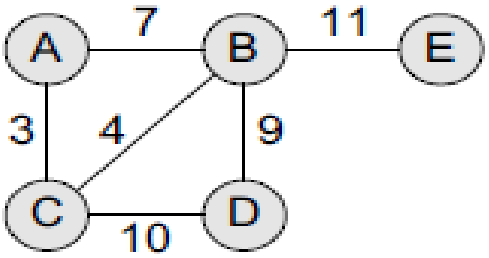
Step 4: Add the selected edge and the vertex to the minimum spanning tree T
[END OF LOOP]

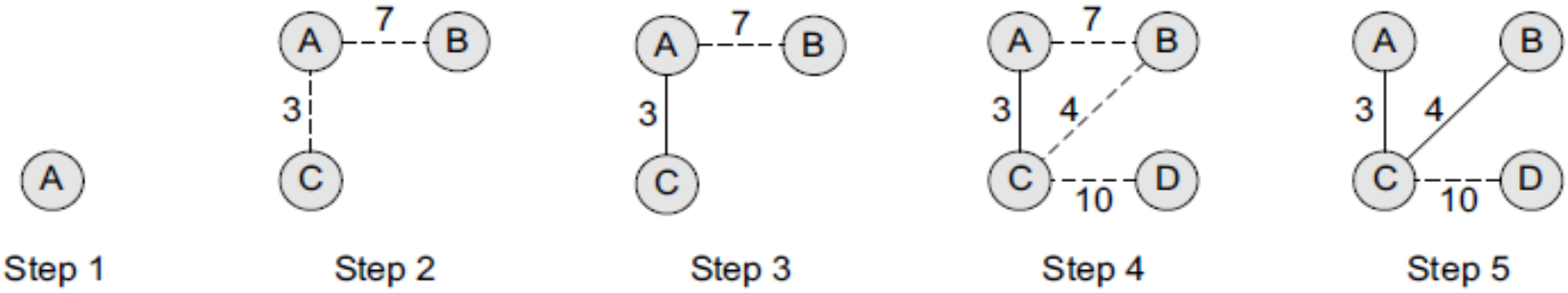
Step 5: EXIT

GRAPHS

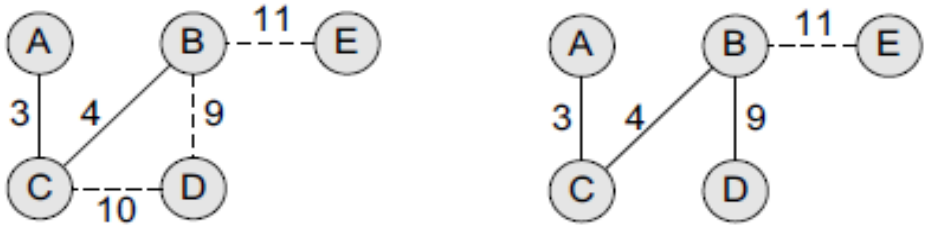
Minimum spanning tree : Prim's Algorithm

Example: Construct a minimum spanning tree of the graph given in Fig. below.





Step 1 Step 2 Step 3 Step 4 Step 5



Step 6 Step 7 Step 8

GRAPHS

Minimum spanning tree : Prim's Algorithm

Step 1: *Choose a starting vertex A.*

Step 2: *Add the fringe vertices (that are adjacent to A). The edges connecting the vertex and fringe* vertices are shown with dotted lines.

Step 3: *Select an edge connecting the tree vertex and the fringe vertex that has the minimum* weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting A and C has less weight, add C to the tree. Now C is not a fringe vertex but a tree vertex.

Step 4: *Add the fringe vertices (that are adjacent to C).*

GRAPHS

Minimum spanning tree : Prim's Algorithm

Step 5: *Select an edge connecting the tree vertex and the fringe vertex that has the minimum* weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting C and B has less weight, add B to the tree. Now B is not a fringe vertex but a tree vertex.

Step 6: *Add the fringe vertices (that are adjacent to B).*

Step 7: *Select an edge connecting the tree vertex and the fringe vertex that has the minimum* weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting B and D has less weight, add D to the tree. Now D is not a fringe vertex but a tree vertex.

Step 8: *Note, now node E is not connected, so we will add it in the tree because a minimum* spanning tree is one in which all the n nodes are connected with n–1 edges that have minimum weight.

GRAPHS

Minimum spanning tree : Kruskal's Algorithm

Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph.

The algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized. However, if the graph is not connected, then it finds a *minimum spanning forest. Note that a forest is a collection of trees. Similarly, a minimum* spanning forest is a collection of minimum spanning trees.

GRAPHS

Minimum spanning tree : Kruskal's Algorithm

Kruskal's algorithm

Step 1: Create a forest in such a way that each graph is a separate tree.

Step 2: Create a priority queue Q that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY

Step 4: Remove an edge from Q

Step 5: IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree).

ELSE

Discard the edge

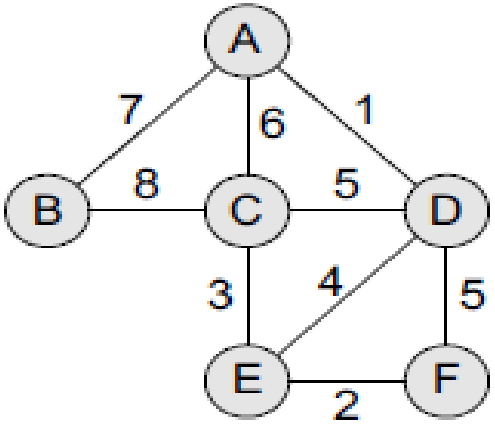
Step 6: END

In the algorithm, we use a priority queue Q in which edges that have minimum weight takes a priority over any other edge in the graph.

GRAPHS

Minimum spanning tree : Kruskal's Algorithm

Example: Apply Kruskal's algorithm on the graph given in fig. below.



Initially, we have F = {{A}, {B}, {C}, {D}, {E}, {F}}
MST = {}
Q = {(A, D), (E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 1: Remove the edge (A, D) from Q and make the following changes:

F = {{A, D}, {B}, {C}, {E}, {F}}
MST = {A, D}
Q = {(E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

GRAPHS

Minimum spanning tree : Kruskal's Algorithm

Step 2: Remove the edge (E, F) from Q and make the following changes:

F = {{A, D}, {B}, {C}, {E, F}}
MST = {(A, D), (E, F)}
Q = {(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 3: Remove the edge (C, E) from Q and make the following changes:

F = {{A, D}, {B}, {C, E, F}}
MST = {(A, D), (C, E), (E, F)}
Q = {(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 4: Remove the edge (E, D) from Q and make the following changes:

F = {{A, C, D, E, F}, {B}}
MST = {(A, D), (C, E), (E, F), (E, D)}
Q = {(C, D), (D, F), (A, C), (A, B), (B, C)}

GRAPHS

Minimum spanning tree : Kruskal's Algorithm

Step 5: Remove the edge (C, D) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST. Therefore,

F = {{A, C, D, E, F}, {B}}
MST = {(A, D), (C, E), (E, F), (E, D)}
Q = {(D, F), (A, C), (A, B), (B, C)}

Step 6: Remove the edge (D, F) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

F = {{A, C, D, E, F}, {B}}
MST = {(A, D), (C, E), (E, F), (E, D)}
Q = {(A, C), (A, B), (B, C)}

GRAPHS

Minimum spanning tree : Kruskal's Algorithm

Step 7: Remove the edge (A, C) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

F = {{A, C, D, E, F}, {B}}
MST = {(A, D), (C, E), (E, F), (E, D)}
Q = {(A, B), (B, C)}

Step 8: Remove the edge (A, B) from Q and make the following changes:

F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {(B, C)}

Step 9: The algorithm continues until Q is empty. Since the entire forest has become one tree, all the remaining edges will simply be discarded. The resultant MST can be given as shown below.

GRAPHS

Minimum spanning tree : Kruskal's Algorithm

F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {}