UNIT 4

4.1 INTRODUCTION

A **graph** is a mathematical structure consisting of a set of vertices (also called nodes) $\{v_1,v_2,\ldots,v_n\}$ and a set of edges $\{e_1,e_2,\ldots,e_n\}$. An edge is a pair of vertices $\{v_i,v_j\}$ $i,j \in \{1\ldots n\}$. The two vertices are called the edge *endpoints*. Graphs are ubiquitous in computer science.

Formally: G = (V, E), where V is a set and $E \subseteq V \times V$

They are used to model real-world systems such as the Internet (each node represents a router and each edge represents a connection between routers); airline connections (each node is an airport and each edge is a flight); or a city road network (each node represents an intersection and each edge represents a block). The wireframe drawings in computer graphics are another example of graphs.

A graph may be either *undirected* or *directed*. Intuitively, an undirected edge models a "two-way" or "duplex" connection between its endpoints, while a directed edge is a one-way connection, and is typically drawn as an arrow. A directed edge is often called an *arc*. Mathematically, an undirected edge is an unordered pair of vertices, and an arc is an ordered pair. The maximum number of edges in an undirected graph without a self-loop is $n(n - 1)/2$ while a directed graph can have at most n^2 edges

G = (V, E) undirected if for all v, w ∈ V: (v, w) ∈ E ⟺ (w, v) ∈ E.
Otherwise directed.For eg.

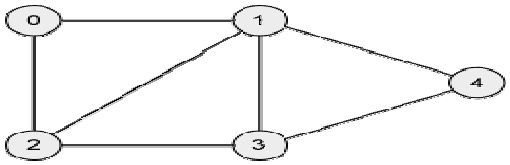


Fig1:Undirect Graph

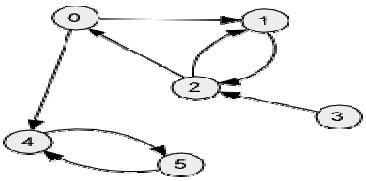


Fig2:Direct Graph

Graphs can be classified by whether or not their edges have **weights**. In **Weighted graph**, edges have a weight. Weight typically shows cost of traversing

Example: weights are distances between cities

In **Unweighted graph**, edges have no weight. Edges simply show connections.

4.2 TERMINOLOGY

- A **graph** consists of:

  A set, V, of **vertices** (nodes)

- A collection, E, of pairs of vertices from V called **edges** (arcs)

  Edges, also called arcs, are represented by **(*u*, *v*)** and are either:

  Directed if the pairs are ordered **(*u*, *v*)**
  u the **origin**
  v the **destination**

- **Undirected End-vertices** of an edge are the **endpoints** of the edge.
- Two vertices are **adjacent** if they are endpoints of the same edge.
- An edge is **incident** on a vertex if the vertex is an endpoint of the edge.
- **Outgoing edges** of a vertex are directed edges that the vertex is the origin. **Incoming edges** of a vertex are directed edges that the vertex is the destination.
- **Degree** of a vertex, *v*, denoted *deg*(*v*) is the number of incident edges. **Out-degree**, *outdeg*(*v*), is the number of outgoing edges. **In-degree**, *indeg*(*v*), is the number of incoming edges.
- **Parallel edges** or multiple edges are edges of the same type and end-vertices **Self-loop** is an edge with the end vertices the same vertex **Simple graphs** have **no** parallel edges or self-loops
- **Path** is a sequence of alternating vetches and edges such that each successive vertex is connected by the edge. Frequently only the vertices are listed especially if there are no paralleledges.
- **Cycle** is a path that starts and end at the same vertex.
- **Simple path** is a path with distinct vertices.
- **Directed path** is a path of only directed edges
- **Directed cycle** is a cycle of only directed edges.
- **Sub-graph** is a subset of vertices and edges.
- **Spanning sub-graph** contains all the vertices.
- **Connected graph** has all pairs of vertices connected by at least one path.
- **Connected component** is the maximal connected sub-graph of a unconnected graph.
- **Forest** is a graph without cycles.
- **Tree** is a connected forest (previous type of trees are called rooted trees, these are free trees)
- **Spanning tree** is a spanning sub graph that is also a tree.

4.3 GRAPH REPRESENTATIONS

There are two standard ways of maintaining a graph G in the memory of a computer.

1. The sequential representation
2. The linked representation

4.3.1SEQUENTIAL REPRESENTATION OF GRAPHS

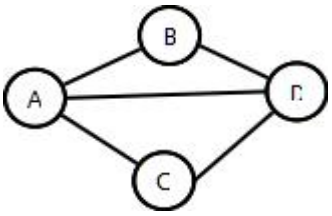
Adjacency Matrix Representation

An **adjacency matrix** is one of the two common ways to represent a graph. The adjacency matrix shows which nodes are **adjacent** to one another. Two nodes are adjacent if there is an edge connecting them. In the case of a directed graph, if node j is adjacent to node i, there is an edge from i to j . In other words, if j is adjacent to i, you can get from i to j by traversing one edge. For a given graph with n nodes, the adjacency matrix will have dimensions of nxn. For an unweighted graph, the adjacency matrix will be populated with Boolean values.

For any given node i, you can determine its adjacent nodes by looking at row (i,[1…n]) adjacency matrix. A value of true at (i,j) indicates that there is an edge from node i to node j, and false indicating no edge. In an undirected graph, the values of (i,j) and (j,i)will be equal. In a weighted graph, the boolean values will be replaced by the weight of the edge connecting the two nodes, with a special value that indicates the absence of an edge.

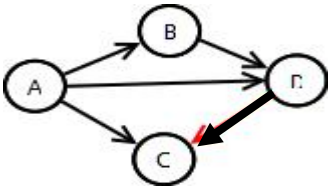
The memory use of an adjacency matrix is $O(n^2)$.

- Example undirected graph (assume self-edges not allowed):



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

- Example directed graph (assume self-edges allowed):



	A	B	C	D
A	∞	1	1	1
B	∞	∞	∞	1
C	∞	∞	∞	∞
D	∞	∞	1	∞

4.3.2 LINKED LIST REPRESENTATION OF GRAPH

Adjacency List Representation

The adjacency list is another common representation of a graph. There are many ways to implement this adjacency representation. One way is to have the graph maintain a list of lists, in which the first list is a list of indices corresponding to each node in the graph. Each of these refer to another list that stores a the index of each adjacent node to this one. It might also be useful to associate the weight of each link with the adjacent node in this list.

Example: An undirected graph contains four nodes 1, 2, 3 and 4. 1 is linked to 2 and 3. 2 is linked to 3. 3 is linked to 4.

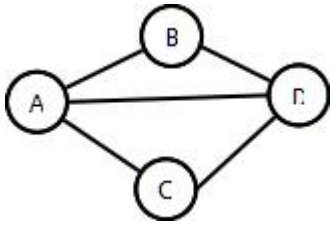
1 - [2, 3]

2 - [1, 3]

3 - [1, 2, 4]

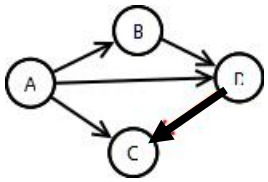
4 - [3]

- Example (undirected graph):



  - A: B, C, D
  - B: A, D
  - C: A, D
  - D: A, B, C

- Example (directed graph):



  - A: B, C, D
  - B: D
  - C: Nil
  - D:C

Adjacency Multi-lists

Adjacency Multi-lists are an edge, rather than vertex based, graph representation. In the Multi-list representation of graph structures; these are two parts, a directory of Node information and a set of linked list of edge information. There is one entry in the node directory for each node of

the graph. The directory entry for node i points to a linked adjacency list for node i. each record of the linked list area appears on two adjacency lists: one for the node at each end of the represented edge.

Typically, the following structure is used to represent an edge.

visited	vertex at tail	vertex at head	Pointer to next edge containing vertex at tail	Pointer to next edge containing vertext at head

Every graph can be represented as list of such EDGE NODEs. The node structure is coupled with an array of head nodes that point to the first edge that contains the vertex as it tail, i.e., the first entry in the ordered pair.

i.e. The node structure in Adjacency multi-list can be summarized as follows:

<M, V1, V2, Link1, Link2> where

M: Mark,

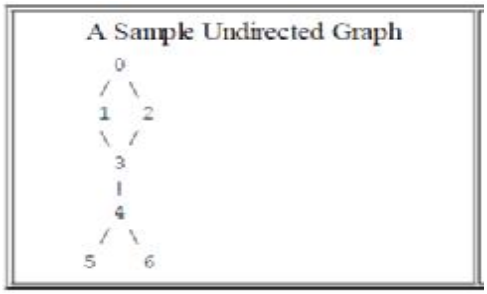
V1: Source Vertex of Edge,

V2: Destination Vertex of Edge,

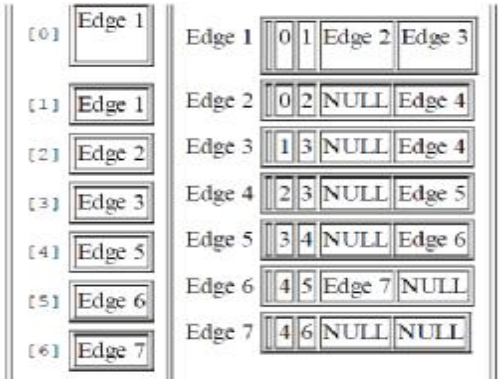
Link1: Address of other node (i.e. Edge) incident on V1,

Link2: Address of other node (i.e. Edge) incident on V2.

Let us consider the following sample UNDIGRAPH:



The adjacency multi-list for the above UNIGRAPH would be as follows:



4.4 GRAPH TRAVERSAL

Graph traversal is the problem of visiting all the nodes in a graph in a particular manner, updating and/or checking their values along the way. The order in which the vertices are visited may be important, and may depend upon the particular algorithm.

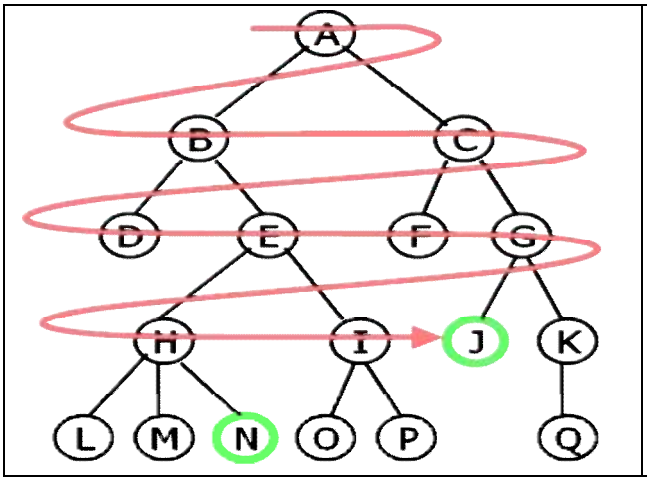
The two common traversals:

breadth-first

depth-first

Breadth First Search

The bread-first-search algorithm starts at a vertex i and visits, first the neighbours of i, then the neighbours of the neighbours of i, then the neighbours of the neighbours of the neighbours of i, and so on. This algorithm is a generalization of the breadth-first traversal algorithm for binary trees. It uses a queue.



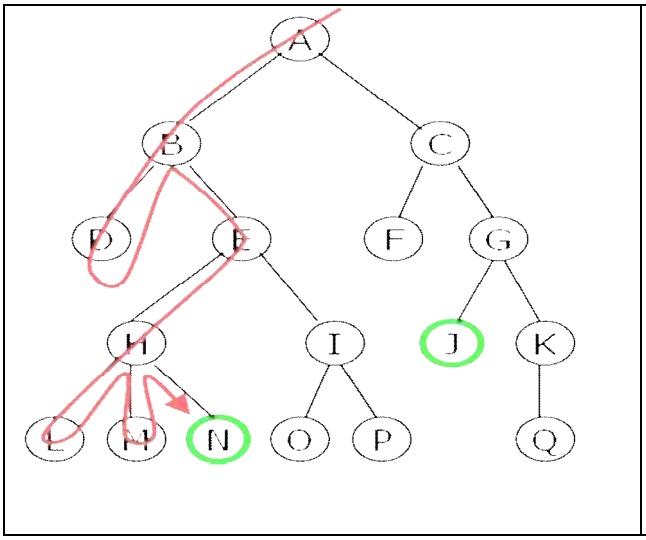
- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N

Algorithm_BFS

1. *Initialize an array A and a queue Q.*
2. *Read the vertex Vi from which you want to start the traversal.*
3. *Initialize the index of the array equal to 1 at the index of Vi*
4. *Insert the vertex Vi into the queue Q*
5. *Visit the vertex which is at the front of the queue. Delete it from the queue and place its adjacent nodes in the queue(if at that index in the array the entry is not equal to 1)*
6. *Repeat step 5 till the queue Q is empty*

Depth First Search

The depth-first-search algorithm is similar to the standard algorithm for traversing binary trees; it first fully explores one subtree before returning to the current node and then exploring the other subtree. Another way to think of depth-first-search is by saying that it is similar to breadth-first search except that it uses a stack instead of a queue.



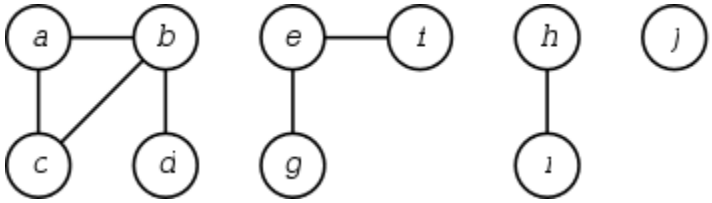
- Eg. A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K
- N will be found before J

Algorithm_DFS

1. *Initialize an array A and a stack S.*
2. *Read the vertex Vi from which you want to start the traversal.*
3. *Initialize the index of the array equal to 1 at the index of Vi*
4. *Insert the vertex Vi into the Stack S*
5. *Visit the vertex which is at the top of the stack. Delete it from the queue and place its adjacent nodes in the stack(if at that index in the array the entry is not equal to 1)*
6. *Repeat step 5 till the queue Q is empty.*

4.5 CONNECTED COMPONENT

A connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.



For example, the graph shown in the illustration above has four connected components {a,b,c,d}, {e,f,g}, {h,i}, and {j}. A graph that is itself connected has exactly one connected component, consisting of the whole graph.

4.6 SPANNING TREE

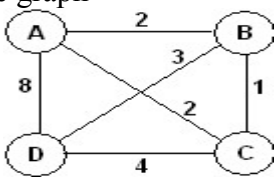
A spanning tree of a graph, G, is a set of **|V|**-1 edges that connect all vertices of the graph.Thus a minimum spanning tree for G is a graph, T = (V', E') with the following properties:

- V' = V
- T is connected
- T is acyclic.

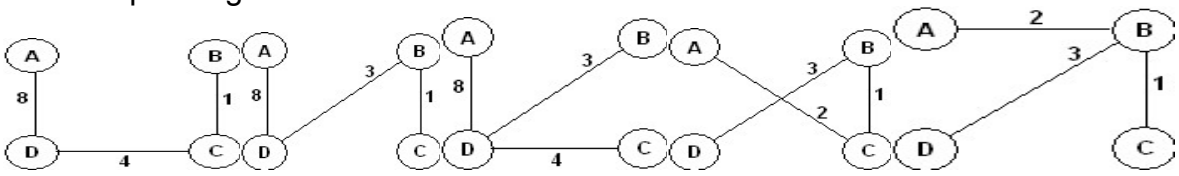
Minimum Spanning Tree

In general, it is possible to construct multiple spanning trees for a graph, G. If a cost, Cij, is associated with each edge, Eij = (Vi,Vj), then the minimum spanning tree is the set of edges, Espan, forming a spanning tree, such that:

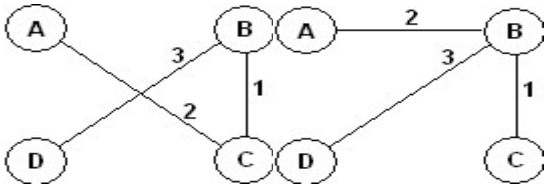
C = sum(Cij | all Eij in Espan) is a minimum.
Eg. The graph



Has 16 spanning trees. Some are:



The graph has two minimum-cost spanning trees, each with a cost of 6:



Minimum Spanning Trees : why do need it?

Suppose we have a group of islands and we wish to link them with bridges so that it is possible to travel from one island to any other in the group. Further suppose that (as usual) our government wishes to spend the absolute minimum amount on this project (because other factors

like the cost of using, maintaining, etc, these bridges will probably be the responsibility of some future government). The engineers are able to produce a cost for a bridge linking each possible pair of islands. The set of bridges which will enable one to travel from any island to any other at minimum capital cost to the government is the minimum spanning tree.

There are two basic Algorithm regarding Minimum Spaning Tree

-*Kruskal's Algorithm* and

-*Prim's Algorithm*

4.6.1 Kruskal's Algorithm

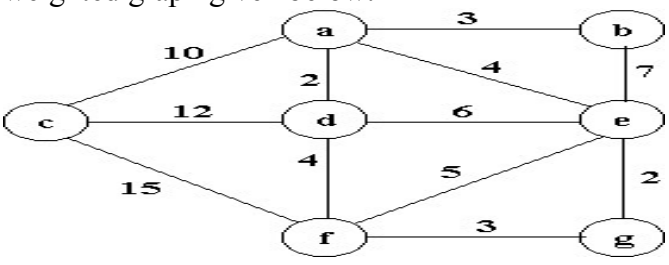
Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Algorithm

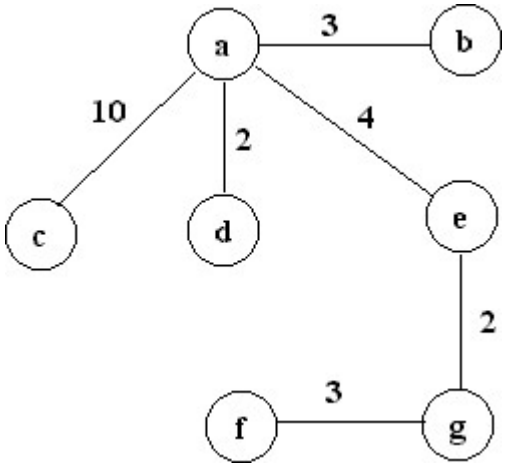
1. *create a forest F (a set of trees), where each vertex in the graph is a separate tree*
2. *create a set S containing all the edges in the graph*
3. *while S is nonempty and F is not yet spanning*
4. *remove an edge with minimum weight from S*
5. *if that edge connects two different trees, then add it to the forest, combining two trees into a single tree*

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree.

Eg:Trace Kruskal's algorithm in finding a minimum-cost spanning tree for the undirected, weighted graph given below:



edge	ad	eg	ab	fg	ae	df	ef	de	be	ac	cd	cf
weight	2	2	3	3	4	4	5	6	7	10	12	15
insertion status	√	√	√	√	√	x	x	x	x	√	x	x
insertion order	1	2	3	4	5					6		



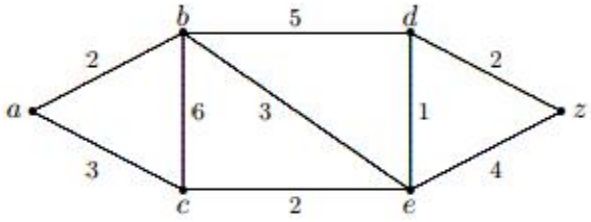
Therefore The minimum cost is: 24

4.6.2 Prim's Algorithm

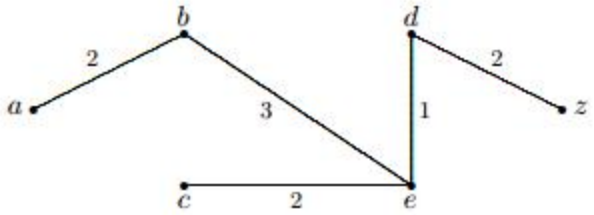
The *Prim's* algorithm makes a nature choice of the cut in each iteration – it grows a single tree and adds a light edge in each iteration.

Algorithm

1. *Initialize a tree with a single vertex, chosen arbitrarily from the graph.*
2. *Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.*
3. *Repeat step 2 (until all vertices are in the tree).*
   Eg.: Use Prim's algorithm to find a minimum spanning tree in the following weighted graph. Use alphabetical order to break ties.



Solution: Prim's algorithm will proceed as follows. First we add edge {d, e} of weight 1. Next, we add edge {c, e} of weight 2. Next, we add edge {d, z} of weight 2. Next, we add edge {b, e} of weight 3. And finally, we add edge {a, b} of weight 2. This produces a minimum spanning tree of weight 10. A minimum spanning tree is the following.



Difference between Kruskal's and Prim's Algorithm

Prim's algorithm initializes with a node whereas Kruskal algorithm initializes with an edge.

Kruska's builds a minimum spanning tree by adding one edge at a time. The next line is always the shortest (minimum weight) ONLY if it does NOT create a cycle.

Prims builds a mimimum spanning tree by adding one vertex at a time. The next vertex to be added is always the one nearest to a vertex already on the graph.

In prim's algorithm, graph must be a connected graph while the Kruskal's can function on disconnected graphs too.

4.7 TRANSITIVE CLOSURE AND SHORTEST PATH ALGORITHM

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 transitive closure of a graph. The graph is given in the form of adjacency matrix say 'graph[V][V]' where graph[i][j] is 1 if there is an edge from vertex i to vertex j or i is equal to j, otherwise graph[i][j] is 0.

Main idea: a path exists between two vertices i, j, iff

there is an edge from i to j; or

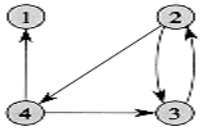
there is a path from i to j going through intermediate vertices which are drawn from set {vertex 1}; or

there is a path from i to j going through intermediate vertices which are drawn from set {vertex 1, 2}; or

there is a path from i to j going through intermediate vertices which are drawn from set {vertex 1, 2, … k-1}; or

there is a path from i to j going through intermediate vertices which are drawn from set {vertex 1, 2, … k}; or

there is a path from i to j going through any of the other vertices



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Shortest Path algorithm

The shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

This is analogous to the problem of finding the shortest path between two intersections on a road map: the graph's vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of its road segment. The Minimal Spanning Tree problem is to select a set of edges so that there is a path between each node. The sum of the edge lengths is to be minimized.

The Shortest Path Tree problem is to find the set of edges connecting all nodes such that the sum of the edge lengths from the root to each node is minimized.

4.7.1 Dijikstra Algorithm

Djikstra's algorithm solves the problem of finding the shortest path from a point in a graph (the *source*) to a destination. It turns out that one can find the shortest paths from a given source to *all* points in a graph in the same time, hence this problem is sometimes called the single-source shortest paths problem.

The somewhat unexpected result that *all* the paths can be found as easily as one further demonstrates the value of reading the literature on algorithms!

This problem is related to the spanning tree one. The graph representing all the paths from one vertex to all the others must be a spanning tree - it must include all vertices. There will also be no cycles as a cycle would define more than one path from the selected vertex to at least one other vertex. Steps of the algorithm are

1. Initial

Select the root node to form the set S1. Assign the path length 0 to this node. Put all other nodes in the set S2.

2. Selection

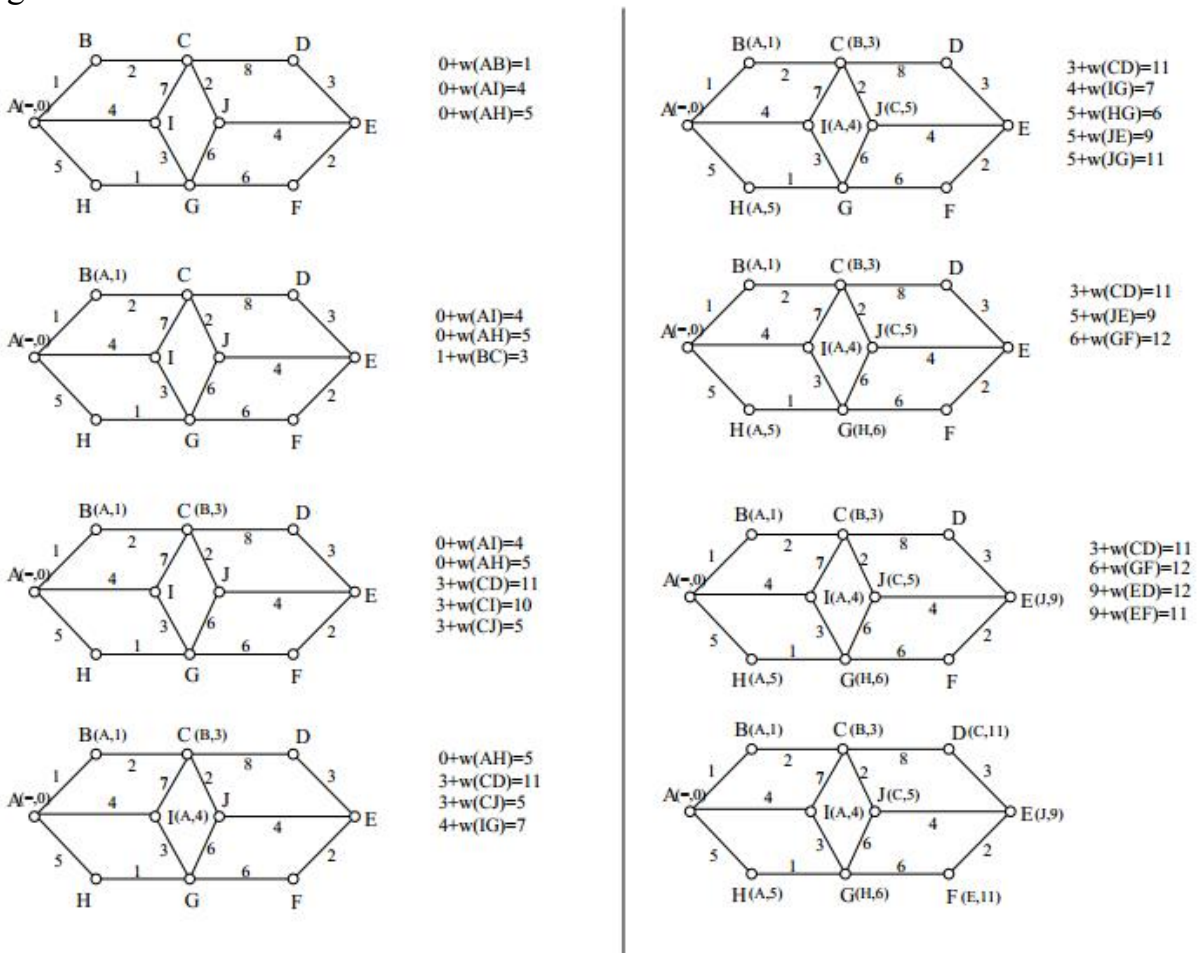
   Compute the lengths of the paths to all nodes directly reachable from S1 through a node in S1. Select the node in S2 with the smallest path length.

   Let the edge connecting this node with S1 be (i, j). Add this edge to the shortest path tree. Add node j to the set S1 and delete it from the set S2.

3. Finish

   If the set S1 includes all the nodes, stop with the shortest path tree. Otherwise repeat the Selection step.

   For
   Eg:



4.7.2 Warshall's Algorithm

Warshall's Algorithm is a graph analysis algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles, see below) and also for finding transitive closure of a relation R.

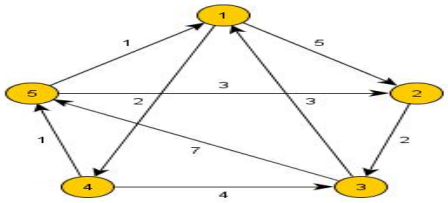
Floyd-Warshall algorithm uses a matrix of lengths D_0 as its input. If there is an edge between nodes i and j, than the matrix D_0 contains its length at the corresponding coordinates. The diagonal of the matrix contains only zeros. If there is no edge between edges i and j, than the position (i,j) contains positive infinity. In other words, the matrix represents lengths of all paths between nodes that does not contain any intermediate node.

In each iteration of Floyd-Warshall algorithm is this matrix recalculated, so it contains lengths of paths among all pairs of nodes using gradually enlarging set of intermediate nodes. The matrix D_1, which is created by the first iteration of the procedure, contains paths among all nodes using exactly one (predefined) intermediate node. D_2 contains lengths using two predefined intermediate nodes. Finally the matrix D_n uses n intermediate nodes.

This transformation can be described using the following recurrent formula:

$$D_{ij}^{n} = \min(D_{ij}^{n-1}, D_{ik}^{n-1} + D_{kj}^{n-1})$$

For Eg.

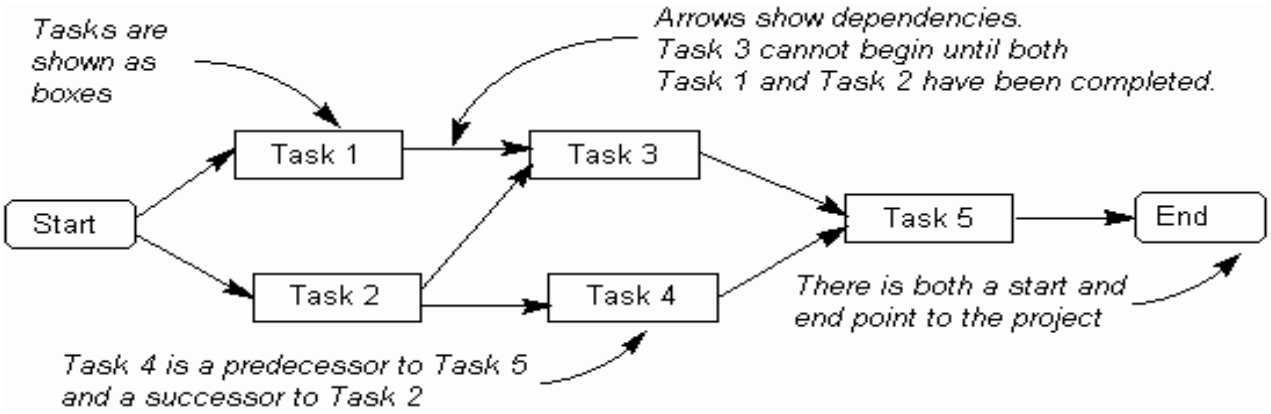


$$D_0 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & 3 & 0 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0 & 5 & 7 & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 7 & 2 & 14 \\ 5 & 0 & 2 & 7 & 9 \\ 3 & 8 & 0 & 5 & 7 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_4 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_5 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 2 & 1 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

4.8 INTRODUCTION TO ACTIVITY NETWORKS

An activity network is a directed graph in which the vertices represent activities and edges represent precedence relation between the tasks. The activity network is also called Activity On Vertex (AOV) network.



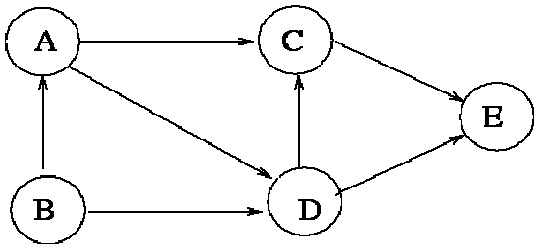
Topological sort gives the order in which activities can be performed.

Step 1. choose the node having indegree = 0.

Step 2. remove that node and edges connecting to it.

Step 3. Goto step 1.

Eg:



Topological sort will give the following sequence of activities

B->A->D->C->E

UNIT-5