Chapter 10: Graphs (5 hours)

Topics to focus on:

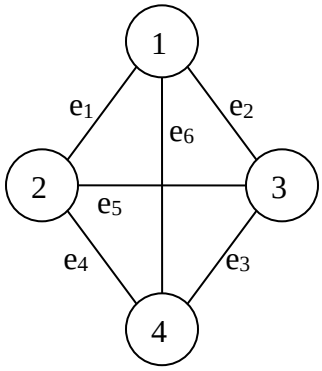
- Introduction
- Graphs as an ADT
- Transitive Closure and Warshall's Algorithm
- Types of graph
- Graph Traversal
- Round robin and Krushal algorithm
- Dijkstra's Algorithm

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

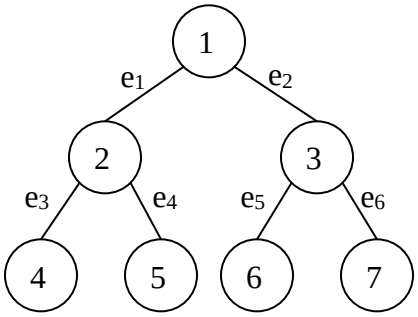
Definition: A graph G = (V, E) consists of a finite set of non-empty set of vertices V and set of edges E.

$V = \{ v_1, v_2, \ldots, v_n\}$
$E = \{e_1, e_2, \ldots e_n\}$
Each edge e is a pair (v, w) where v, w ϵ V. The edge is also called arc.

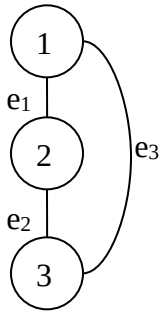
Example of graphs:



G_1



G_2



G_3

$V(G_1) = \{1, 2, 3, 4\}$
$E(G_1) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$
$e_1 = (1, 2)$
$e_2 = (1, 3)$
$e_3 = (3, 4)$
$e_4 = (2, 4)$
$e_5 = (2, 3)$
$e_6 = (1, 4)$

$V(G_3) = \{1, 2, 3\}$
$E(G_3) = \{e_1, e_2, e_3\}$
$e_1 = (1, 2)$
$e_2 = (2, 3)$
$e_3 = (1, 3)$

Now, specify the vertices and edges for G_2.

The vertices are represented by points or circles and the edges are line segments connecting the vertices. If the graph is directed, then the line segments have arrow heads indicating the direction.

Applications of Graphs:

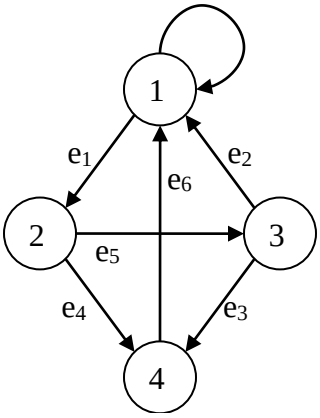
1. **Social networks**: A social network is by definition, well, a network. And graphs are special cases of networks, with only a single type of edge between vertices.
2. **Web graphs**: The web is a huge collection of documents pointing to each other via hyperlinks. In other words, the web is another massive graph data set.
3. **Networks**: The environment is actually one of the largest sources of real-world graphs. Examples are brain networks, protein interaction networks, food networks and neural network in AI.
4. **Road networks:** Apps like Maze, Google Maps, Apple Maps, and Uber are installed on every smartphone. Navigational problems are inherently modeled as graph problems. Think about the traveling salesman problem, shortest path problems, Hammington paths, etc.
5. **Bitcoin transaction graphs:** The Blockchain is an interesting graph that is often analyzed in the cryptocurrency space. Another insightful graph arises when you use Bitcoin wallets as vertices and transactions between wallets as edges. The resulting graph reflects the money flow between Bitcoin wallets. This graph is critical to learning about global money flow patterns.

Types of Graphs:

1. Directed Graphs
2. Undirected Graphs
3. Weighted Graphs
4. Simple Graph
5. Null graph
6. Complete graph

1. Directed Graphs:

If every edge (i, j), in E(G) of a graph G is marked by a direction from i to j, then the graph is called directed graph.



G_4

In edge e = (i, j), we say, e leaves i and enters j. In directed graphs, self loops are allowed. The indegree of a vertex v is the number of edges entering v. The outdegree of a vertex v is the number of edges leaving v.

2. Undirected Graph:

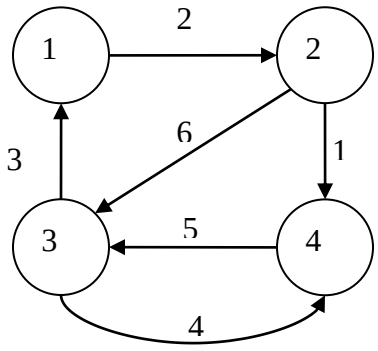
If the directions are not marked for any edge, the graph is called undirected graph. The graphs G_1, G_2, G_3 are undirected graphs. In an undirected graph, we say that an edge e = (u, v) is incident on u and v (u and v are connected).

Undirected graph doesn't have self-loops.

3. Weighted Graphs:

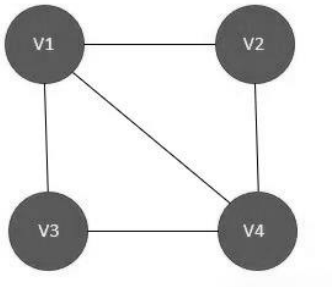
A graph is said to be weighted graph if every edge in the graph is assigned some weight or value. The weight is a positive value that may represent the cost of moving along the edge, distance between the vertices etc.

The two vertices with no edge (path) between them can be thought of having an edge (path) with weight infinite.



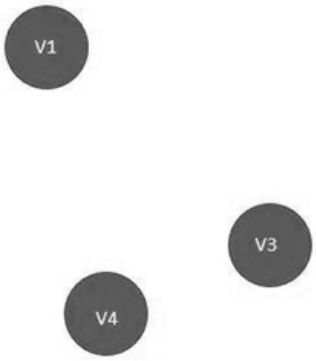
4. Simple Graph:

A graph G= (V, E) is said to be a simple graph if there is one and only one edge between each pair of vertices. Thus, there is only edge connecting 2 vertices and can be used to show one to one relationship between 2 elements.



5. Null Graph:

A graph G= (V, E) is said to a null graph if there are n number of vertices exist, but no Edge exists that connects then.



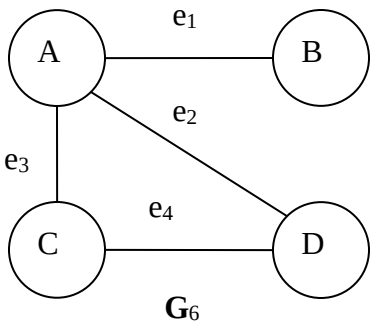
6. Complete Graph:

A *complete graph* is a graph in which there is an edge between every pair of vertices.

Additional Terminology in Graph

Adjacent and Incident

Two vertices i and j are called adjacent if there is an edge between the two. Example. The vertices adjacent to vertex A, in the fig below are B, C, D. The adjacent vertices of C are A and D.



G_6

If e(i, j) is an edge on E(G), then we say that the edge e(i, j) is incident on vertices i and j. Eg. in the above figure, e_4 is incident on C and D. If (i, j) is directed edge, then i is adjacent to j and j is adjacent from i.

Path:

A path is a sequence or distinct vertices, eadh adjacent to the next. For eg. the sequence of vertices e_1, e_3, e_4 (i.e (B,A), (A,C), (C,D)) of the above graph form a path from B to D. The length of the path is the number of edges in the path. So, the path from B to D has length equal to three.

In a weighted graph, the cost of a path is the sum of the costs of its edges. Loops have path length of 1.

Cycle:

A cycle is a path containing at least three vertices such that the last vertex on the path is adjacent to the first. In the above graph G_6, A, C, D, A is a cycle.

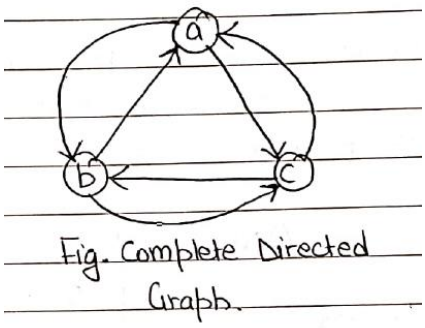
Connected:

Any graph is *connected* provided there exists a path (directed or undirected) between any two nodes.

Strongly Connected:

A graph is said to be **strongly connected** if every pair of vertices (u, v) in the graph contains a path between each other.

In an unweighted directed graph G, every pair of vertices u and v should have a path in each direction between them i.e., bidirectional path.



Representation of Graphs

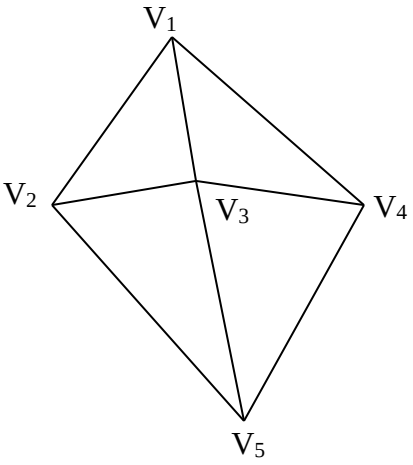
A graph can be represented in many ways. Some of them are described in this section.

1. Adjacency matrix

Adjacency matrix A for a graph G = (V, E) with n vertices, is n x n matrix, such that
A_{ij} = 1, if there is an edge from v_i to v_j
A_{ij} = 0, if there is no such edge

We can also write,

$$A(i, j) = \begin{cases} 1 \text{ if and only if } (v_i, v_j) \text{ exists} \\ 0 \text{ otherwise} \end{cases}$$

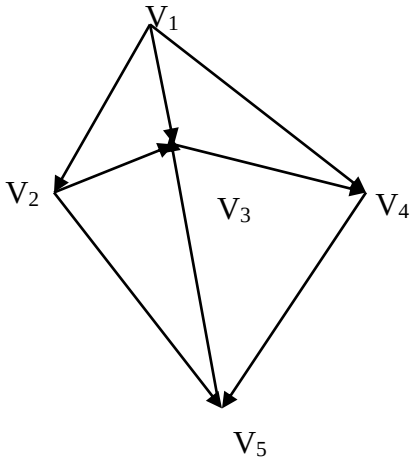


Example: An adjacency matrix for the following undirected graph is

A =

	V1	**V2**	**V3**	**V4**	**V5**
V1	0	1	1	1	0
V2	1	0	1	0	1
V3	1	1	0	1	1
V4	1	0	1	0	1
V5	0	1	1	1	0

If the graph is directed, then the adjacency matrix will be as follows. The number in each row tells the outdegree of that vertex.



A =

	V1	**V2**	**V3**	**V4**	**V5**
V1	0	1	1	1	0
V2	0	0	1	0	1
V3	0	0	0	1	0
V4	0	0	0	0	1
V5	0	0	1	0	0

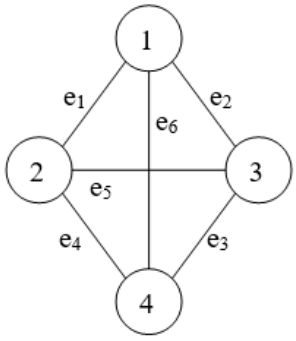
In this representation we require n^2 bits to represent a graph with n nodes. It is a simple way to represent a graph, but it has following disadvantages:

- it takes $O(n^2)$ space
- it takes $O(n^2)$ time to solve most of the problems.

2. Set Representation

Two sets are maintained. They are

i. Set of vertices
ii. Set of edges

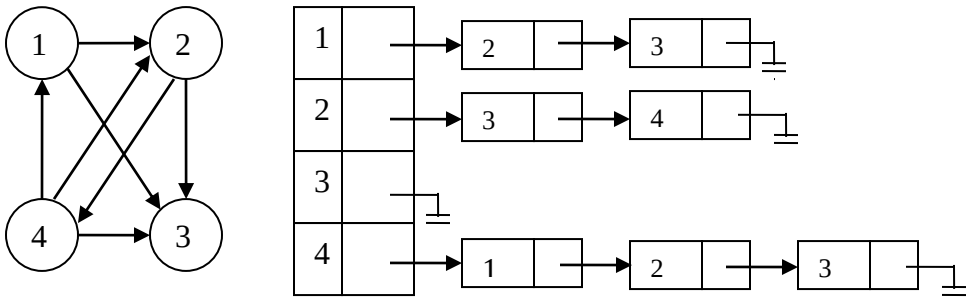


G_1

$V(G_1) = \{1, 2, 3, 4\}$
$E(G_1) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$
$e_1 = (1, 2)$, $e_2 = (1, 3)$, $e_3 = (3, 4)$, $e_4 = (2, 4)$, $e_5 = (2, 3)$, $e_6 = (1, 4)$

3. Adjacency List Representation

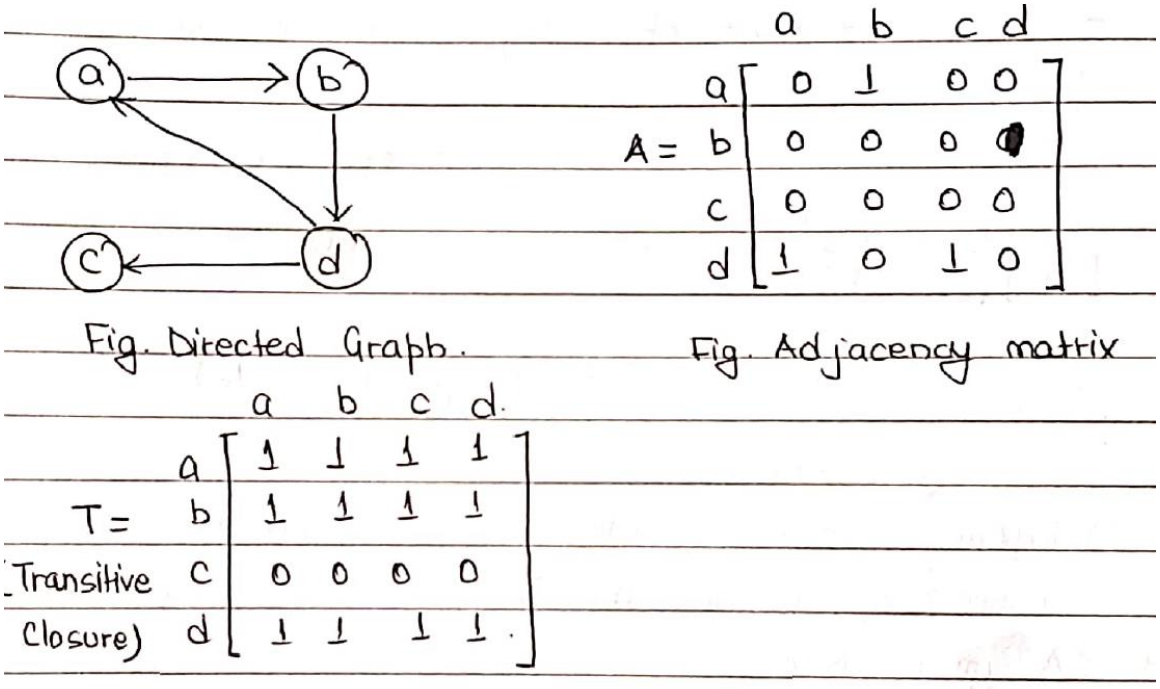
The n rows of an adjacency matrix can be represented as n linked lists. This is one list for each node in a graph. Each list will contain adjacent nodes. Each node has two fields, *Vertex* and *Link*. The Vertex field of a node p will contain the nodes that are adjacent to the node p.



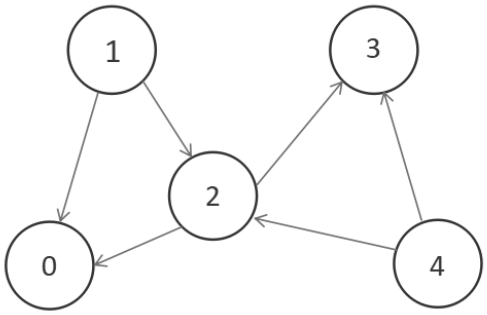
Transitive Closure and Warshall's Algorithm:

In any Directed Graph, let's consider a node i as a starting point and another node j as ending point.

For all (i,j) pairs in a graph, transitive closure matrix is formed by the reachability factor, i.e if j is reachable from i (means there is a path from i to j) then we can put the matrix element as 1 or else if there is no path, then we can put it as 0.



Question: Find out the adjacency matrix and the transitive closure of the graph given below:



Warshall's Algorithm:

Warshall's algorithm is used to determine the transitive closure of a directed graph or all paths in a directed graph by using the adjacency matrix.

In Warshall's algorithm, we construct a sequence of Boolean matrices A= d_0, d_1, d_2……d_n = T, where A is adjacency matrix and T is its transitive closure. This can be done from digraph D as follows:

$[d_1]_{i,j}$ = 1 if and only if there is a path from V_i to V_j with elements of a subset of $\{V_1\}$ as interior vertices.

$[d_2]_{i,j}$ = 1 if and only if there is a path from V_i to V_j with elements of a subset of $\{V_1, V_2\}$ as interior vertices.

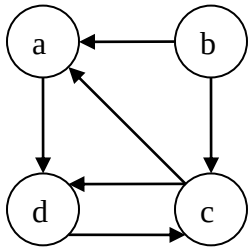
Continuing the process, we generalized to

$[d_n]_{i,j}$ = 1 if and only if there is a path from V_i to V_j with elements of a subset of $\{V_1, V_2, \ldots..V_k\}$ as interior vertices.

Question: Define Diagraph Explain Warshall's algorithm to find the transitive closure of a diagraph.

Example of Warshall's Algorithm

Transitive closure method of finding path between nodes is quite inefficient because it processes the adjacency matrix lots of times. An Efficient method for calculating transitive closure is by Warshall's algorithm whose time complexity is only $O(n^3)$.



We first find the matrices d_0, d_1, d_2, d_3, d_4, and d_4 is the transitive closure.

$$d_0 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad d_1 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1^* \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

d_1 has 1 as its $(i,j)^{th}$ entry if there is a path from vi to vj that has v1 = a as an interior vertex.

$$d_2 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad d_3 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1^* & 0 & 1 & 1^* \end{bmatrix}$$

d2 has 1 as its $(i,j)^{th}$ entry if there is a path from v_i to v_j that has v_1 and v_2 as interior vertex. d_3 has 1 as its $(i,j)^{th}$ entry if there is a path fro v_i to v_j that has v_1 v_2 and v_3 as interior vertex.

$$d_4 = \begin{bmatrix} 1^* & 0 & 1^* & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1^* & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Finally, d_4 has 1 as its $(i,j)^{th}$ entry if there is a path from v_i to v_j that has v_1, v_2, v_3 and v_4 as interior vertex.

The matrix d_4 is the transitive closure.

Warshall's Algorithm

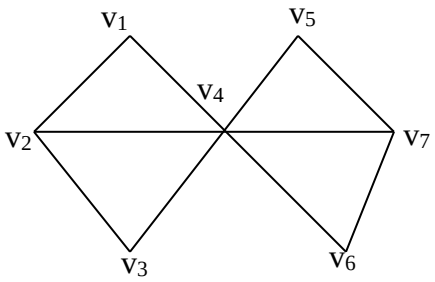
```
procedure warshall (M_R : n x n zero one matrix)
{
        D = M_R
        for k = 1 to n
        begin
          for i = 1 to n
          begin
                for j = 1 to n
                        d_ij = d_ij ∨ (d_ik ∧ d_kj)
        end
}
```

$D = [d_{ij}]$ is the transitive closure.

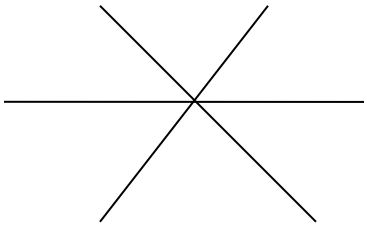
Spanning Tree:

Let a graph G=(V,E) be a graph. If T is a subgraph of G and contains all the vertices but no cycles/circuits, then 'T' is said to be spanning tree of G.

In other words, the spanning tree of an undirected graph G is the free tree formed from graph edges which connects all the vertices of G.

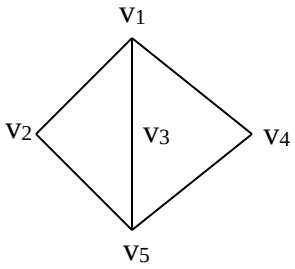


Undirected Graph G

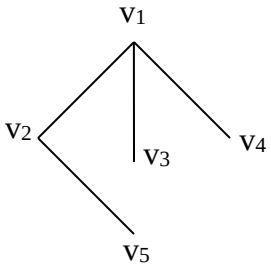


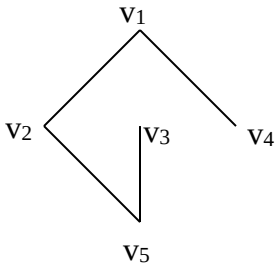
Spanning Tree of G

For a given graph G=(V, E), if Breadth-First search and Depth-First search call is made, then the resultant tree that is generated is the spanning tree of the graph G. Thus, by making a Depth-First search on G, we get the Depth-First spanning tree. Similarly, on applying Breadth-First search on graph 'G' we get Breadth First spanning tree.



Graph G





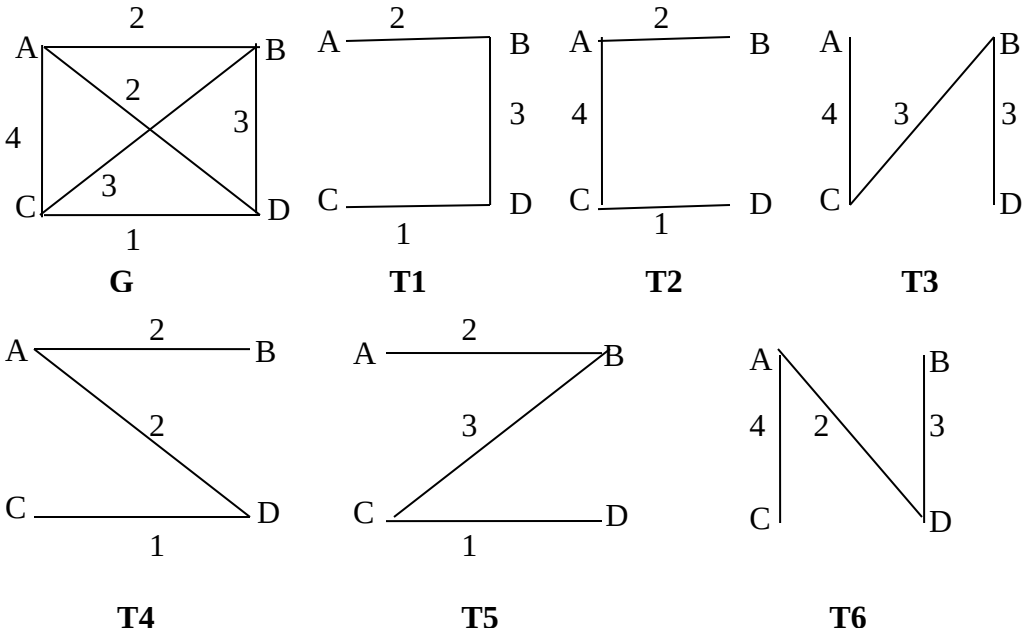
BS Spanning Tree and DF Spanning Tree of G

Minimum Spanning Tree

The cost of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in tree.

The minimum spanning tree of a weighted undirected graph is the spanning tree that connects all the vertices in G at lowest cost. So, if T is the minimum spanning tree of graph G and T' is any other spanning tree of G then,

$$\mathbf{w(T) \leq w(T')}$$



Here, **T_1** to **T_6** are the *spanning trees* of **G**. Among them, **T4** has minimum cost (i. e. 5). So, **T_4** is the minimum spanning tree of graph G

There are several algorithms available to determine the *minimal spanning tree* of a given weighted graph.

GRAPH AS AN ADT

Values: A graph of G (V, E) contains number of vertices and edges.

Operations:

1. Graph (): Creates a new empty graph G.
2. addVertix (Vertex): Add a vertex to the graph G.
3. addEdge (fromVertex, toVertex): Add an edge between the starting and the ending vertex.
4. addEdgewithweight (fromVertex, toVertex, Weight): Add an edge between the starting and the ending vertex with a weight w.
5. getVetices (): Returns all the vertices in the graph.

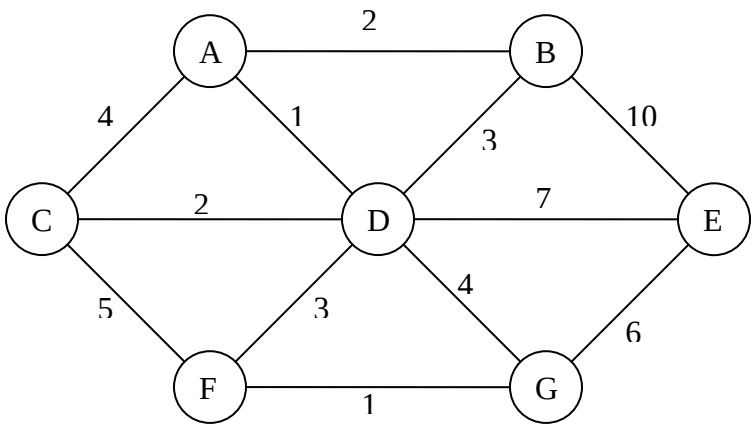
Kruskal's Algorithm

To determine minimum spanning tree, consider a graph G with n vertices.

Step 1: List all the edges of the graph G with increasing weights

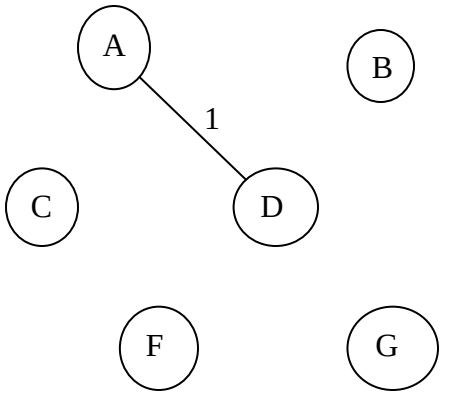
Step 2: Proceed sequentially to select one edge at a time joining n vertices of G such that no cycle is formed until n-1 edges are selected.

Step 3: Draw the n-1 edges that were selected forming a minimal spanning tree T of G.

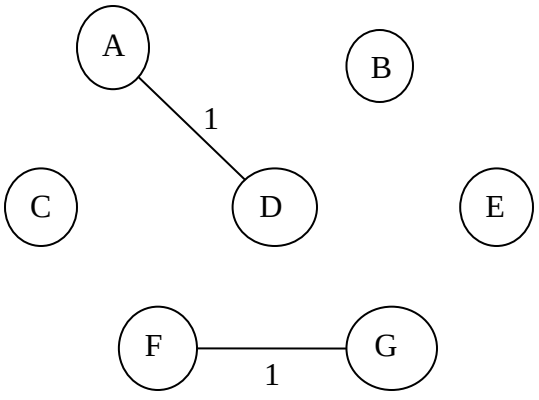


We have,

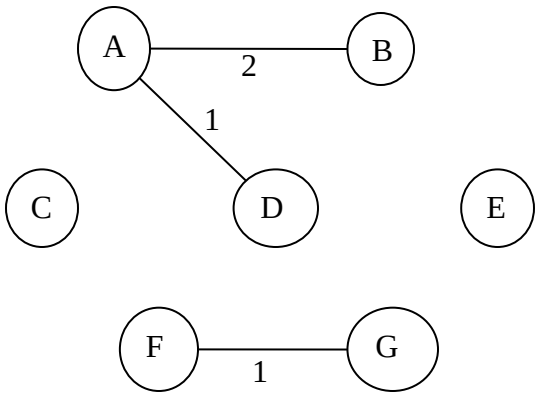
Edges:	AD	FG	AB	CD	BD	AC	DG	CF	EG	DE	DF	BE
Weights	1	1	2	2	3	4	4	5	6	7	8	10



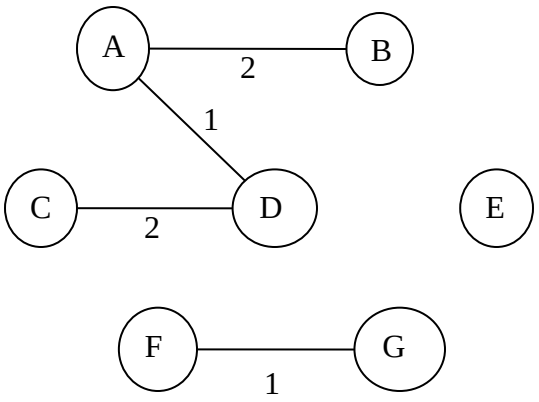
(1)



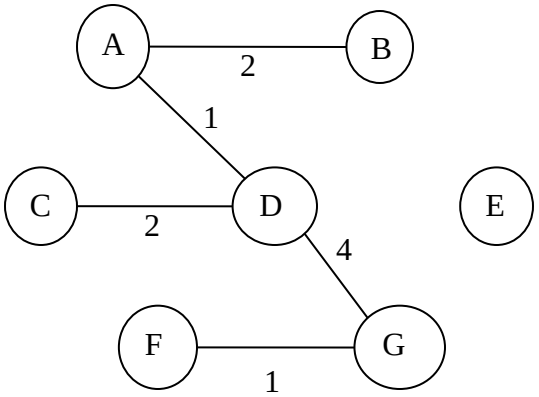
(2)



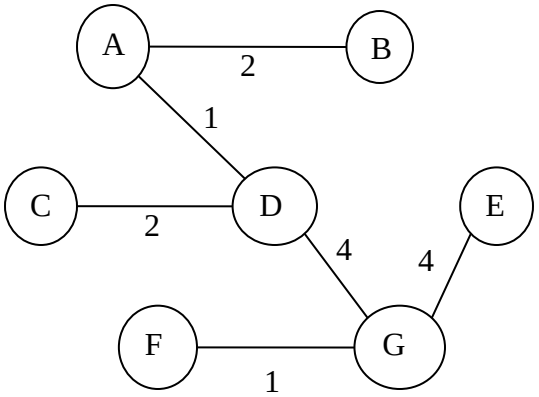
(3)



(4)



(5)



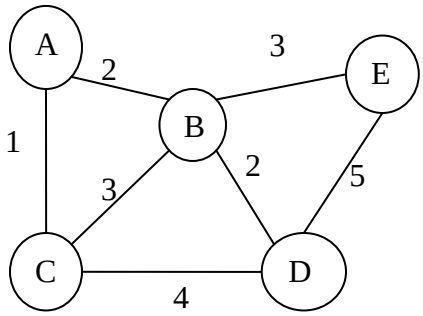
(6)

Hence, the minimum spanning tree is generated.

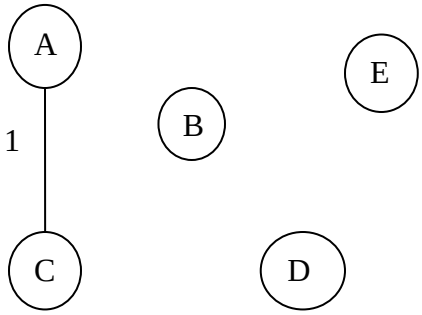
Round Robin Algorithm

This method provides better performance when the number of edges is low. Initially each node is considered to be a partial tree. Each partial tree is maintained in a queue Q. Priority queue is associated with each partial tree, which contains all the arcs ordered by their weights.

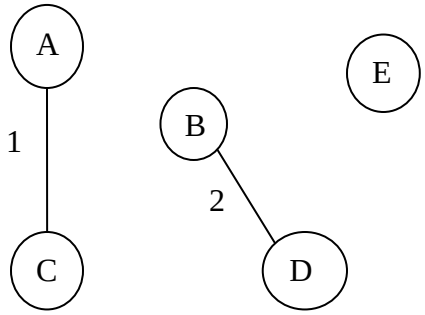
The algorithm proceeds by removing a partial tree, T1, from the front of Q, finding the minimum weight arc a in T1; deleting from Q, the tree T2, at the other end of arc a; combining T1 and T2 into a single new tree T3 and at the same time combining priority queues of T1 and T2 and adding T3 at the rear of priority queue. This continues until Q contains a single tree, the minimum spanning tree.



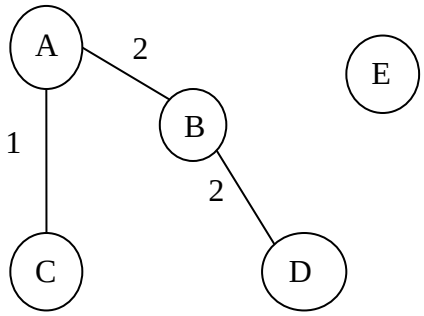
Q	Priority queue
{A}	1, 2
{B}	2, 2, 3, 3
{C}	1, 3, 4
{D}	2, 4, 5
{E}	3, 5



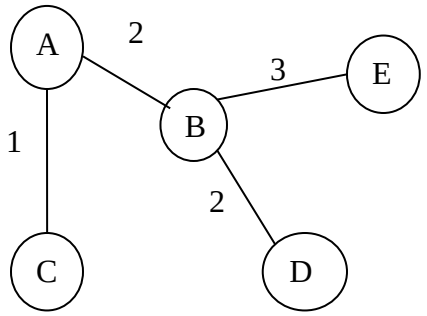
Q	Priority queue
{B}	2, 2, 3, 3
{D}	2, 4, 5
{E}	3, 5
{A, C}	2, 3, 4



Q	Priority queue
{E}	3, 5
{A, C}	2, 3, 4
{B, D}	2, 3, 3, 4, 5



Q	Priority queue
{E}	3,5
{A, C, B, D}	3,3,3,4,4,5



Q	Priority queue
{A, C, B, D, E}	3, 3, 4, 4, 5, 5

Only 1 partial tree is left in the queue, which is the required minimum spanning tree.

Shortest Path Algorithm

There are many problems that can be modeled using graph with weight assigned to their edges. Eg, we may set up the basic graph model by representing cities by vertices and flights by edges. Problems involving distances can be modeled by assigning distances between cities to the edges. Problems involving flight time can be modeled by assigning flight times to edges. Problem involving fares can be modeled by assigning fares to the edges.

Thus, we can model airplane or other mass transit routes by graphs and use shortest path algorithm to compute the best route between two points.

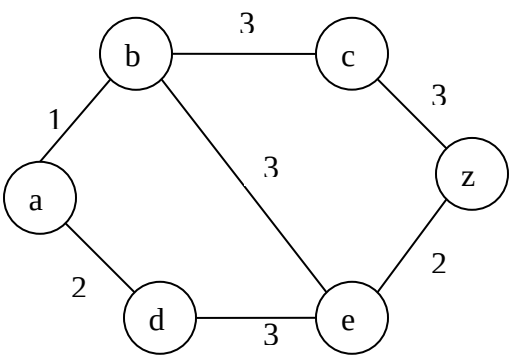
Similarly, if the vertices represent computers; the edges represent a link between computers; and the costs represent communication costs, delay costs, then we can use the shortest-path algorithm to find the cheapest way to send electronic news, data from one computer to a set of other computers.

Basically, there are three types of shortest path problems:

Single path: Given two vertices, s and d, find the shortest path from s to d and its length (weiths).

Single source: Given a vertex, s find the shortest path to all other vertices.

All pairs: Find the shortest path from all pair of vertices.
Let source vertex be a. We will find the shortest path to all the other vertices.

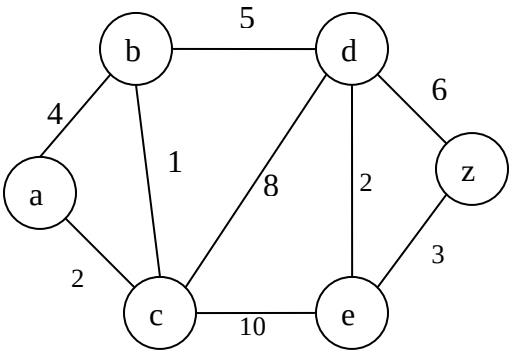


Vertex	Cost	Path
b	1	a, b
c	4	a, b, c
d	2	a, d
e	4	a, b, e
z	6	a, b, e, z

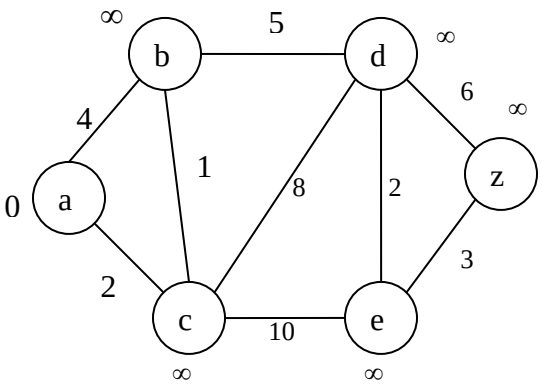
Dijkstra's Algorithm to find the shortest path

1. Mark all the vertices as unknown
2. for each vertex v keep a distance d_v from source vertex s to v initially set to infinity except for s which is set to $d_s = 0$
3. repeat these steps until all vertices are known
    i. select a vertex v, which has the smallest d_v among all the unknown vertices
    ii. mark v as known
    iii. for each vertex w adjacent to v
        i. if w is unknown and $d_v + \text{cost}(v, w) < d_w$
        update d_w to $d_v + \text{cost}(v, w)$

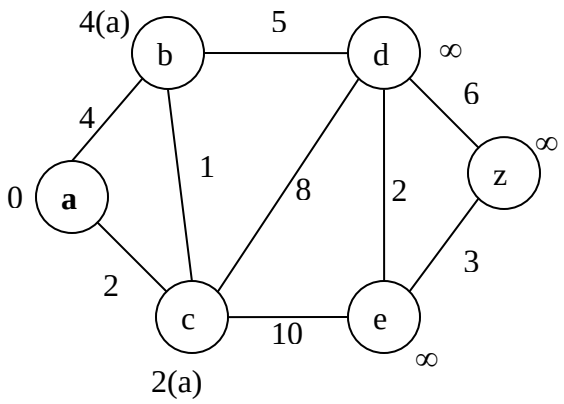
Using Dijkstra's Algorithm to find shortest path from a to z



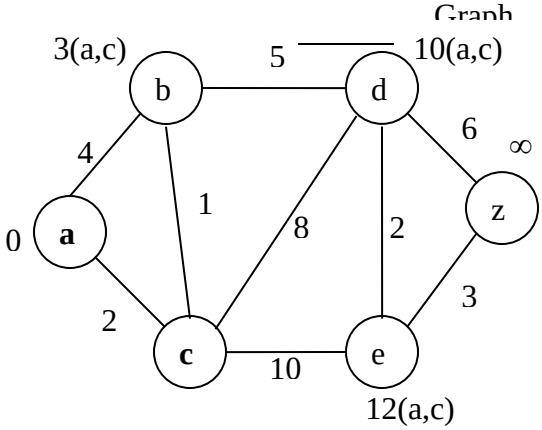
Given graph with weights



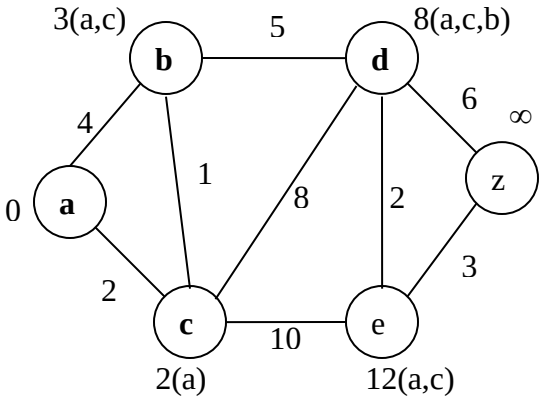
(Step 1)



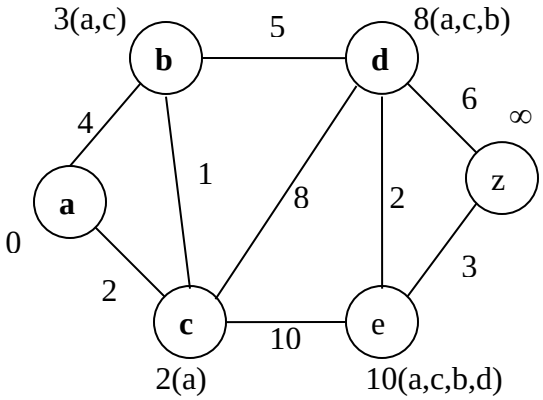
(Step 2)



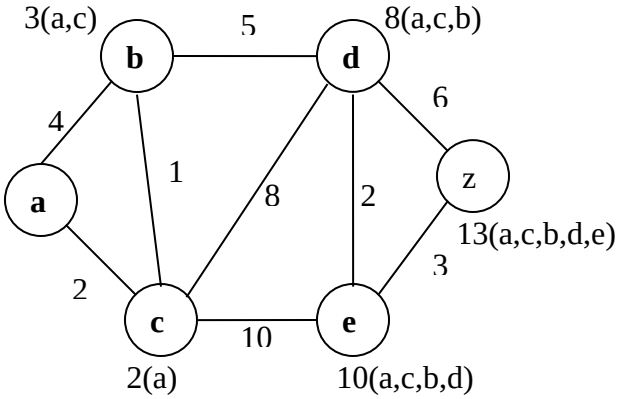
(Step 3)



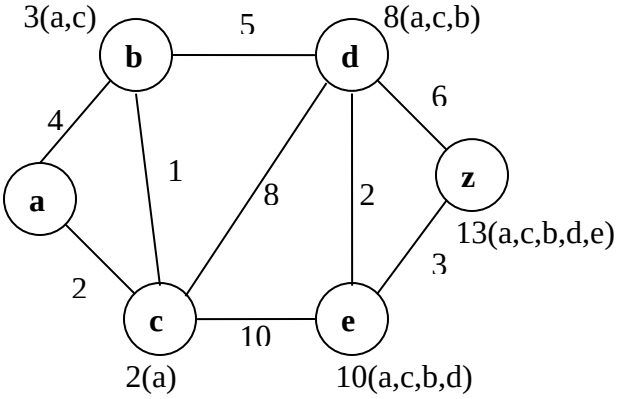
(Step 4)



(Step 5)



(Step 6)



(Step 7)

Fig: Using Dijkstra's Algorithm to find shortest path from a to z

Greedy Algorithm
One of the simplest approaches that often leads to a solution of an optimization problem. This approach selects the best choice at each step, instead of considering all sequences of steps that may lead to an optimal solution. Algorithm that make what seems to be the "best" choice at each step are called greedy algorithm.

Differentiate between Trees and Graphs

TREES	GRAPHS
Tree is a special form of graph i.e. minimally connected graph and having only one path between any two vertices.	In graph, there can be more than one path i.e. graph can have unidirectional or bi-directional paths (edges) between nodes.
Trees cannot have loops, circuits & self loops.	It can have loops, circuits & self loops.
Less complex	More complex
It consists of one root & parent child relationship.	No such concept is present.
Tree is traversed in Pre-Order, In-Order & Post-Order.	Graph is traversed by Depth First Search & Breadth First Search algorithm.
Binary Tree, Binary Search tree, AVL Tree are some of the types of tree.	Directed & Undirected graphs are types of graph.
It is kind of hierarchical model.	It is a type of network model.
Sorting and searching like Tree Traversal & Binary search are the applications of Tree.	Coloring of maps, graph coloring, algorithms, CPM & PERT analysis are the applications of graph.

Traversing in Graph

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

	Breadth First Search.	**Depth First Search.**
Technique	It a vertex-based technique to find the shortest path in a graph.	It is an edge-based technique because the vertices along the edge are explored first from the starting to the end node.
Definition	BFS is a traversal technique in which all the nodes of the same level are explored first, and then we move to the next level.	DFS is also a traversal technique in which traversal is started from the root node and explore the nodes as far as possible until we reach the node that has no unvisited adjacent nodes.
Data Structure	Queue data structure is used for the BFS traversal.	Stack data structure is used for the BFS traversal.
Backtracking	BFS does not use the backtracking concept.	DFS uses backtracking to traverse all the unvisited nodes.
Number of edges	BFS finds the shortest path having a minimum number of edges to traverse from the source to the destination vertex.	In DFS, a greater number of edges are required to traverse from the source vertex to the destination vertex.
Optimality	BFS traversal is optimal for those vertices which are to be searched closer to the source vertex.	DFS traversal is optimal for those graphs in which solutions are away from the source vertex.
Speed	BFS is slower than DFS.	DFS is faster than BFS.
Memory efficient	It is not memory efficient as it requires more memory than DFS.	It is memory efficient as it requires less memory than BFS.