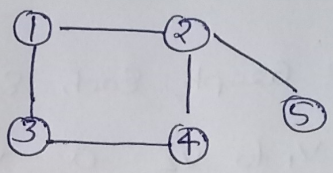
UNIT - 6

GRAPHS

Graph :- A Graph G, is a non-linear data structure of order Pair (V, E). V represents set of vertices and E represents set of edges that connect Pair of vertices

Ex-

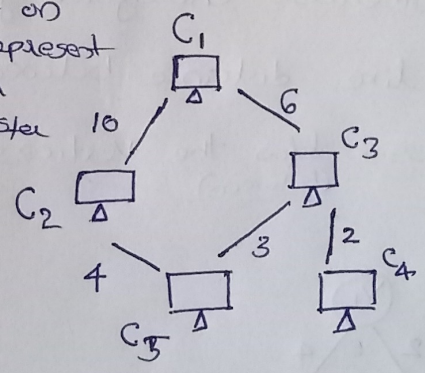


Vertices $V = \{1, 2, 3, 4, 5\}$

Edges $E = \{(1,2), (1,3), (2,4), (3,4), (2,5)\}$

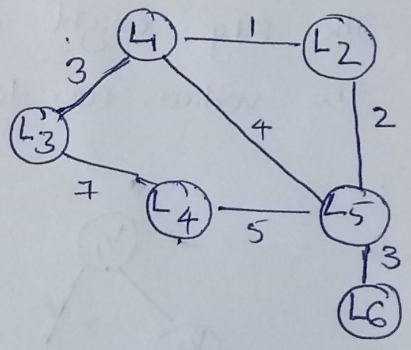
Applications - Graph data structures are widely used in Connected networks. The network can be collection of computers, collection of geographical locations (Google maps)

* weight on Edge represent time for data transfer



- Can we transfer data from C3 to C4 ?
- what is best Path to transfer data from C1 to C5

* weight on Edge represent distance between different locations

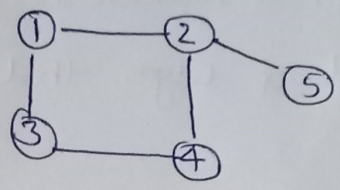


- Can we travel from L3 to L6 ?
- what is best Path to reach from Location L_1 to L5 $L \rightarrow L5$ (a) $L_1 \rightarrow L2 \rightarrow L5$?

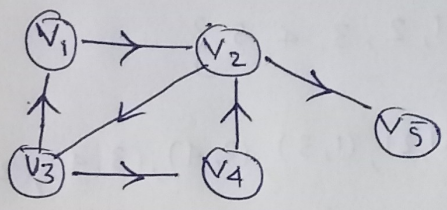
There are several graph algorithms, which Provide solutions to above such Problems. (Here Computer/Location is treated as vertex)

Graphs are commonly categorized into following types

1. Undirected Graphs - The direction of Edge between vertices is not defined, transmission is Possible in both the directions between vertices

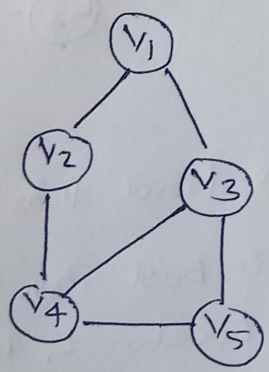


2. Directed Graphs - In the Graph, Each Edge is represented by a direction. Either from V_1 to V_2 or V_2 to V_1

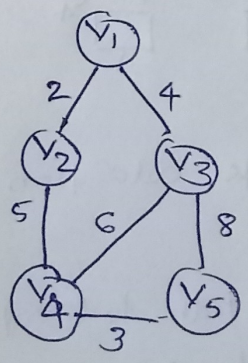


3. Unweighted & Weighted Graphs

If the Edges in a Graph are represented with weights it is known to be weighted graph otherwise unweighted graph The Edge weight can be any metric like distance between the vertices (or) time ~~from~~ for transmission b/w (between) the vertices



Unweighted Graph



weighted Graph

Graph representations

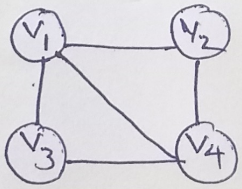
There are two most commonly used graph representation

i) Adjacency Matrix
ii) Adjacency List

As the name suggests we are going store adjacency information, either two vertices have a direct edge (or) not If vertices V_1 and V_2 have an Edge they are said to be adjacent. ~~and E_{12} will be assigned as edge b/w V_1 and V_2.~~

Adjacency Matrix - It is a square matrix of size V, where V is the no of vertices in a Graph. Each cell in the matrix Stores the adjacency between the vertices.

Ex:



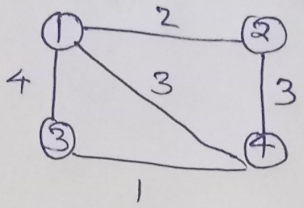
Here Edges = $E_{V_1V_2} = E_{V_2V_1} = 1$
$E_{V_1V_3} = E_{V_3V_1} = 1$
$E_{V_1V_4} = E_{V_4V_1} = 1$
$E_{V_2V_4} = E_{V_4V_2} = 1$
$E_{V_3V_4} = V_4V_3 = 1$

No of vertices = 4. The Adjacency Matrix is AM[4][4], if there is an Edge between vertex i and j assign $AM_{ij} = 1$ otherwise $AM_{ij} = 0$

AM[4][4] =

	V_1	V_2	V_3	V_4
V_1	0	1	1	1
V_2	1	0	0	1
V_3	1	0	0	1
V_4	1	1	1	0

Ex: If the Graph is a weighted Graph, store the Edge weights in the adjacency matrix



AM[4][4] is Adjacency Matrix,

AM_{ij} = Edge weight between vertex i and j

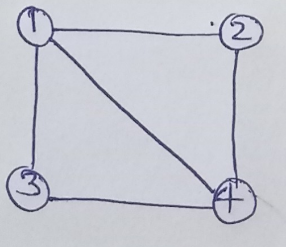
AM_{ij} = ∞ infinity (In Program we will consider any large value like 9999 to represent ∞)

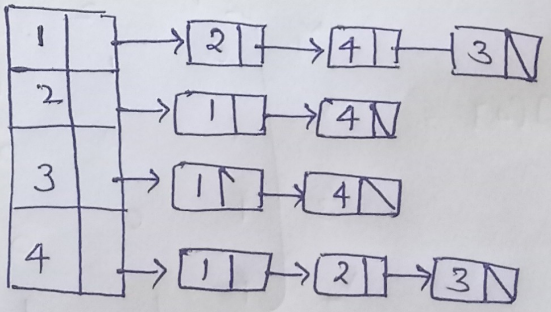
$$AM[4][4] = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} \\ \begin{bmatrix} \infty & 2 & 4 & 3 \\ 2 & \infty & \infty & 3 \\ 4 & \infty & \infty & 1 \\ \infty & 3 & 1 & \infty \end{bmatrix} \end{array} = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} \\ \begin{bmatrix} 9999 & 2 & 4 & 3 \\ 2 & 9999 & 9999 & 3 \\ 4 & 9999 & 9999 & 1 \\ 9999 & 3 & 1 & 9999 \end{bmatrix} \end{array}$$

Adjacency List -

The Linked lists are used for graph representation

For Each vertex, all its adjacent vertices are represented in a Linked List

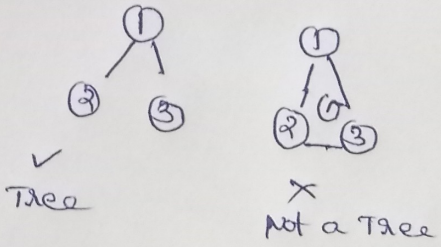


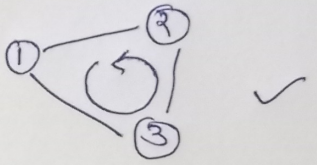


Array of vertices

before getting into Graph algorithms, Lets observe the key differences between Trees and Graphs

	Tree	Graph
①	Root Node → Every Tree must have one Root Node (Exactly one)	Will not have any root, we can select any node as source
②	Loop → Loops cannot be created in Trees	Graph can contain Loops





Graph Traversal techniques –

The Traversal technique will define in which order the vertices in a graph are to be visited. Commonly used Graph Traversal techniques are

i. Breadth First Search (BFS)

ii. Depth First Search (DFS)

Breadth First Search (BFS)

Start traversing the graph by selecting any vertex as a source, visit all its adjacent vertices. Now select First visited adjacent vertex and visit all its neighboring (adjacent) vertices such that they are not yet visited

Algorithm

Step1: Declare queue of size, with no of vertices

Step 2: Select any vertex as Source to start traversal, mark it visited and insert it into the Queue

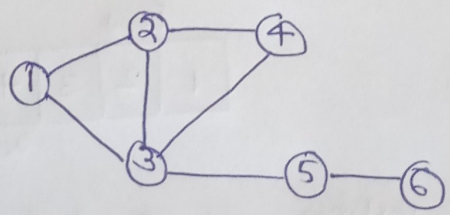
Step 3: Visit all the unvisited adjacent vertices by adding them to queue by taking vertex at Front

Step 4: once all adjacent vertices are added, dequeue (delete) vertex at Front (Front moves to next vertex in queue)

Step 5: Repeat step 3 and 4 until all vertices are dequeue from the queue

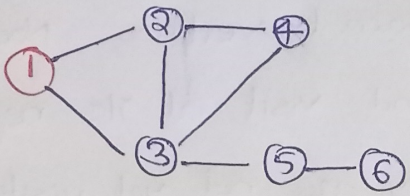
Ex:

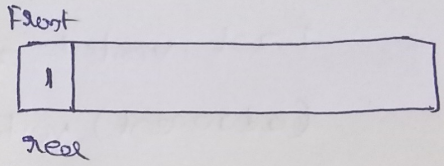
Consider following graph



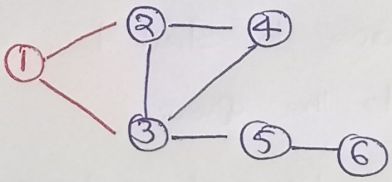
→ Select ① as the source vertex

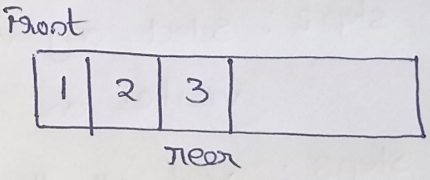
Enqueue ① into the queue



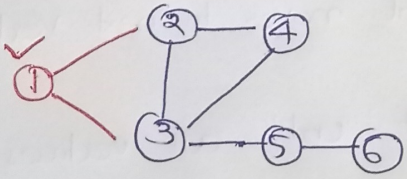


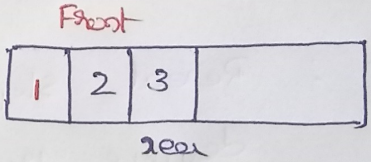
→ Enqueue all adjacent vertices of ① i.e., ② and ③



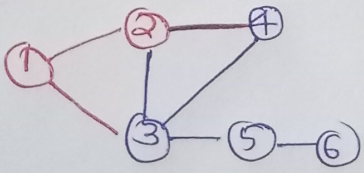


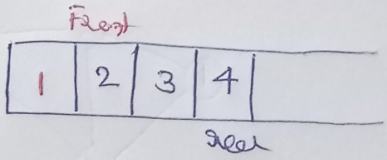
→ ① has no other vertex adjacent to it, so dequeue ① from the queue. Now Front will be incremented





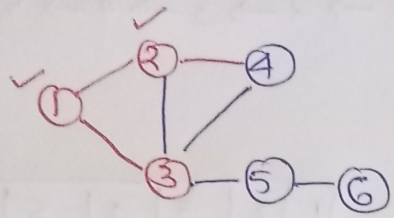
→ Select Element at Front i.e., ② and repeat the above steps

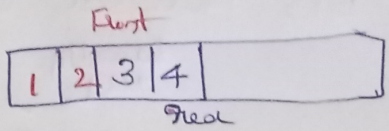




③, ④ are the adjacent vertices of ② but ③ is already visited, in queue so Enqueue ④ only

→ Dequeue Element at Front i.e, ②. Front will be incremented will be at vertex ③ Now

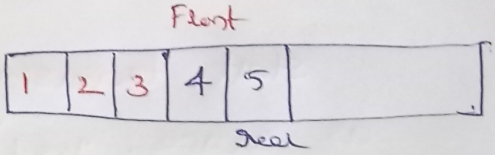




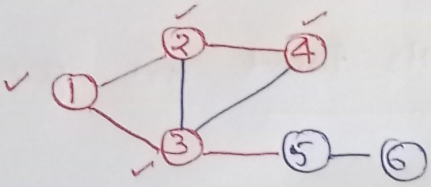
Enqueue adjacent vertices of ③ which are not yet visited, here ①, ②, ④, ⑤ are adjacent but only ⑤ is the unvisited.

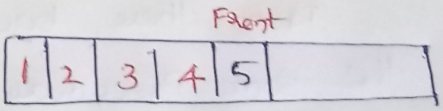
Enque ⑤ and then dequeue ③

→ Now Vertex at Front is ④

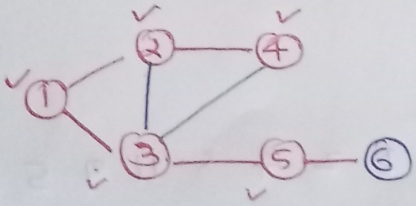


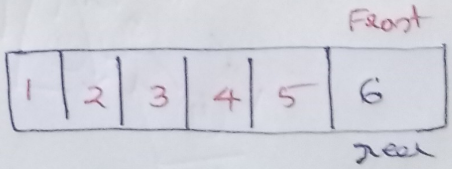
its adjacent vertices are ②, ③
both are already visited so dequeue ④
then Front will be at vertex ⑤





→ ⑤ have an adjacent vertex ⑥ which is not yet visited Enqueue ⑥ to queue then dequeue ⑤. Now Element at Front of queue will be ⑥

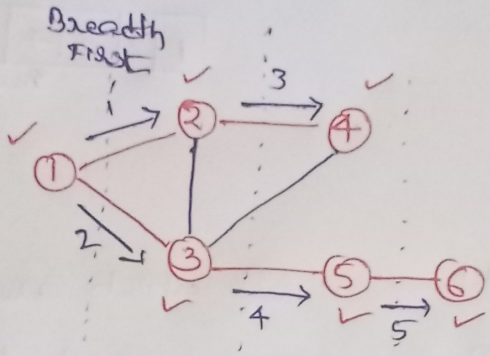


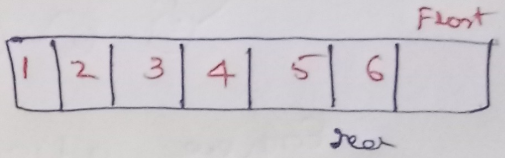


→ ⑥ have no adjacent vertices which are not yet visited so dequeue ⑥. queue is Empty.

when queue becomes Empty. It means BFS traversal is completed.

The order in which Elements are Enqueued/dequeued is the BFS traversal of the Graph 1 → 2 → 3 → 4 → 5 → 6





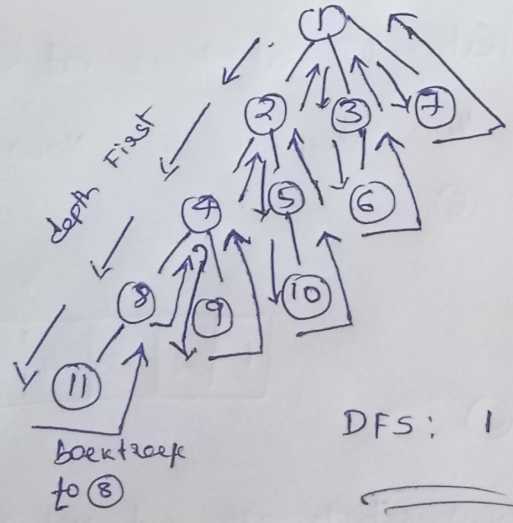
Here Edges ②-③ and ⑤-④ are not visited in BFS

Depth First Search (DFS)

Start traversing the graph by selecting any vertex as source. Visit any unvisited vertex depth wise, from there again visit its unvisited vertex depth wise until we reach last node. (have no child nodes to visit)

From there backtracks to its Parent and again repeat the same Procedure

Ex:



DFS: 1 2 4 8 11 9 5 10 3 6 7

DFS Algorithm

step 1 : Define stack of size with no of vertices in Graph

step 2 : select any vertex as source for traversal. mark its status as visited and Push on to the stack

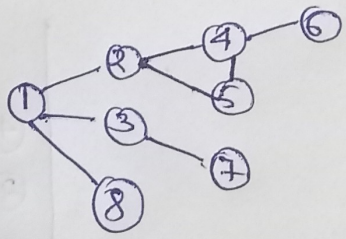
step 3 : visited any of its unvisited adjacent vertices and Push it to the stack

Step 4 : Repeat steps3 untill there is no new vertex to be visited from vertex which is at top of stack

Step 5 : Now Pop the vertex on top of stack, then we will backtrack to its Parent vertex

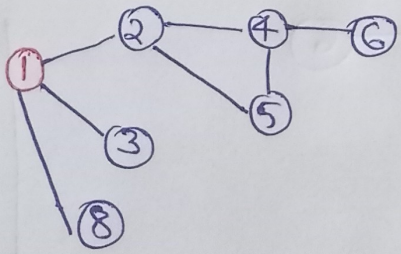
Step 6 : Repeat steps 3, 4 and 5 untill stack becomes Empty

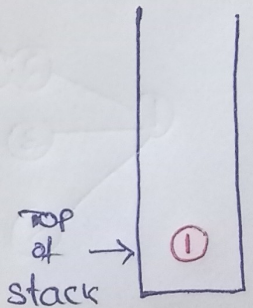
Ex :



Select ① as source vertex,

Set ① as visited and Push into stack

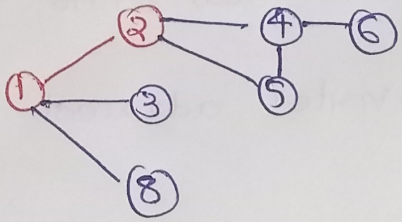


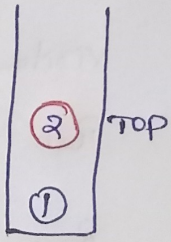


adjacent vertices of ① are ②, ③, ⑧

Select ② and Push it on to the stack

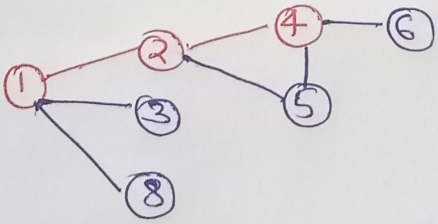
NOTE : we will Comeback and visit ③, ⑧ if they are not visited later

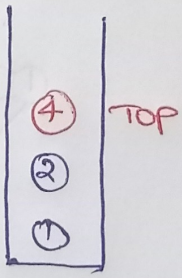




→ TOP of stack is ②, adjacent vertices are ④, ⑤ & ① ① is already visited. From ④ and ⑤ select ④ and Push to stack

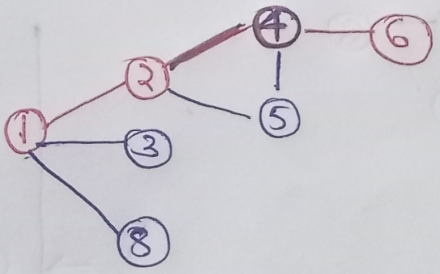
NOTE : we will visit ⑤, if it is not visited later

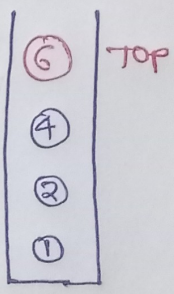




→ TOP of stack is ④ its adjacent vertices are ⑤ and ⑥ Lets Push ⑥ into the stack

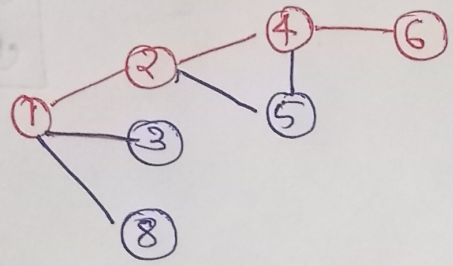
NOTE : ⑤ is not yet visited

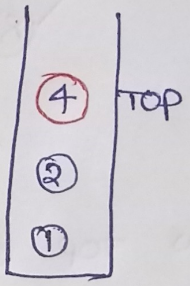




→ Now top of stack is (6), its only adjacent vertex is (4) but it is already visited

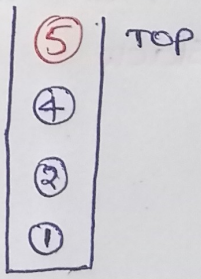
As there no more unvisited vertices from (6) to reach Pop (6) from the stack



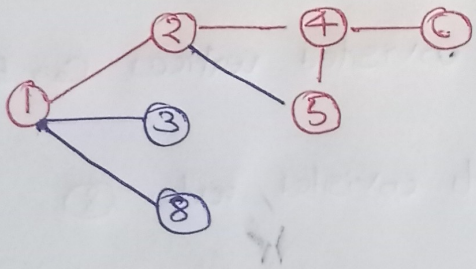


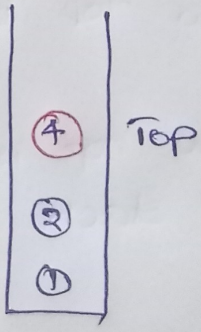
// (6) is Popped from stack

→ Again top of stack is (4), it has Previously unvisited vertex (5) Push it to stack

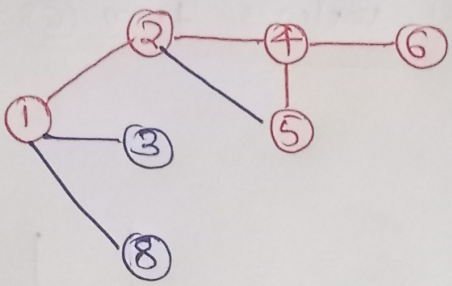


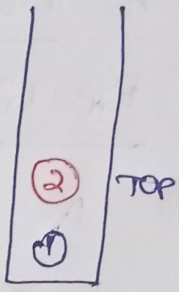
→ Now top of stack is (5), it adjacent vertices are (2) and (4) both are visited no more vertices can be visited from (5), Pop it from stacks





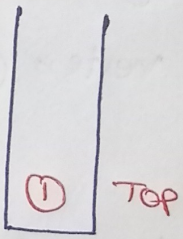
→ Now TOP of stack is ④, All its adjacent vertices are visited So now Pop ④ from stack

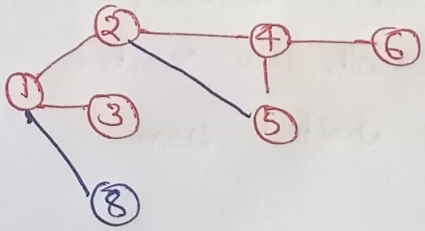


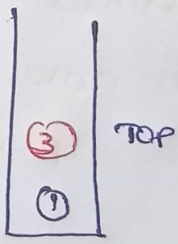


→ Now TOP of stack is ②, its adjacent vertices are ①, ④, ⑤ and all are visited, so Pop ② from stack

→ Now TOP of stack is ①, its unvisited adjacent vertices are ③ & ⑧ Push ③ on stack

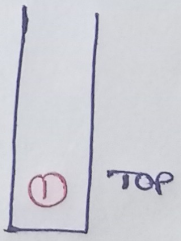


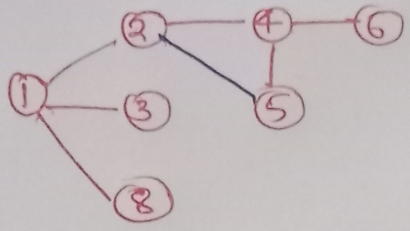


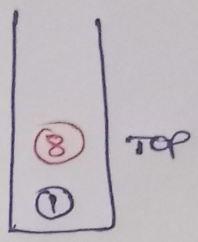


→ Now TOP is ③, but unvisited vertices can be added Pop ③

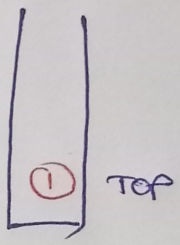
→ Now TOP is ①, Push its unvisited vertex ⑧ to stack



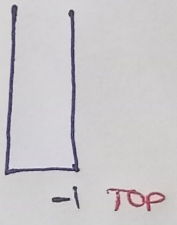




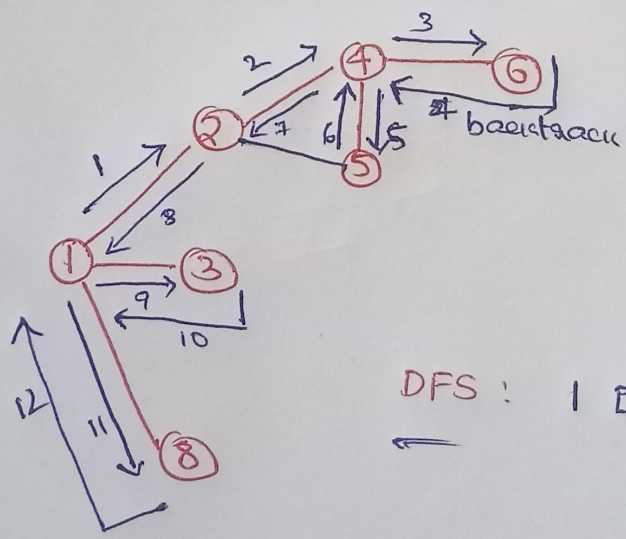
→ Now TOP is ⑧, where we cannot add any new vertex so Pop ⑧ from stack



→ Now TOP is ①, all its adjacent vertices are visited Pop ① from stack



→ Now TOP is -1, means all vertices are visited in DFS order. Traversal is complete

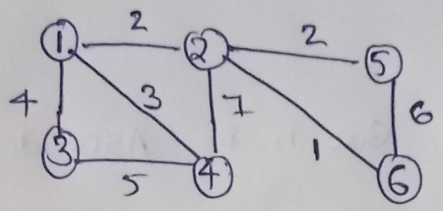


DFS : 1 2 4 6 5 3 8

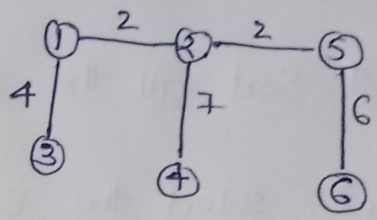
Spanning Tree

A Spanning Tree is the subset of Graph G, which covers all the vertices of graph with minimal number of Edges

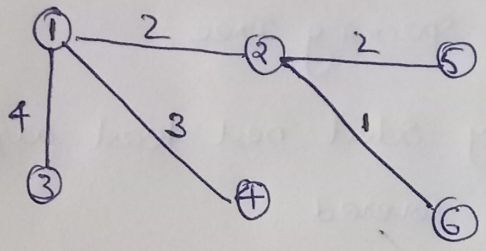
Ex:



Graph



Spanning Tree T_1



Spanning Tree T_2

→ For a Graph G there can exists more than one spanning Tree

→ A spanning Tree will have n-1 Edges, for a graph with n vertices

Minimal Spanning Tree

The spanning Tree which has minimum cost is known to be minimal spanning Tree

cost = Sum of all Edge weights

In above Graph T_2 is a minimal spanning Tree & T_1 is not

$$\text{cost}(T_1) = 4+2+7+2+6 = 21$$

$$\text{cost}(T_2) = 4+2+3+2+1 = 12$$

Thus by using Minimal spanning Tree we can reach all Vertices in less distance/time

The most commonly used algorithms to Construct Spanning Tree are
(i) Kruskal's Algorithm
(ii) Prims Algorithm

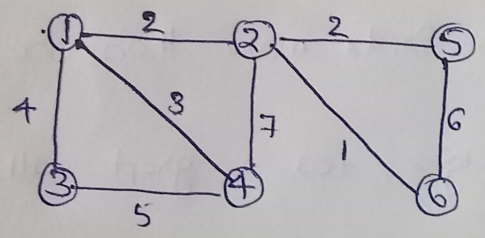
Kruskals Algorithm

Step 1: Sort all the Edges of Graph in Ascending order

Step 2: Select the Edge with least weight and add it to Spanning Tree. If adding the Edges forms a loop then do not add Such Edge to Spanning Tree.

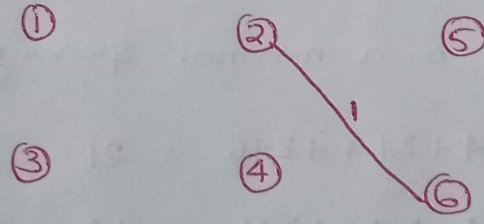
Step 3: Repeat the above step by Select next least weight Edge until all the vertices are Covered

Ex:

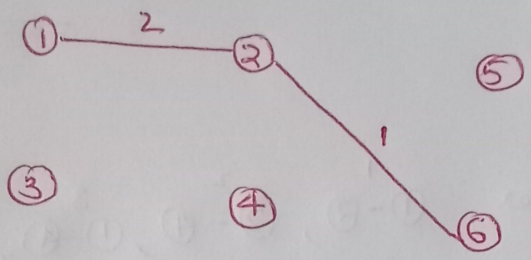


Edge weights in Ascending order { (2,6), (1,2), (2,5), (1,4), (1,3), (3,4), (5,6), (2,4) }
1 2 2 3 4 5 6 7

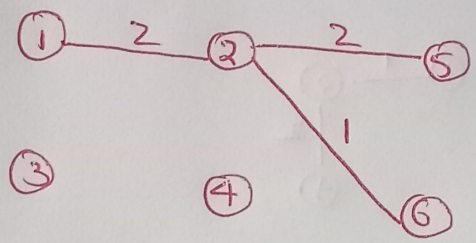
→ Select Edge with least weight (2,6) add to Spanning Tree



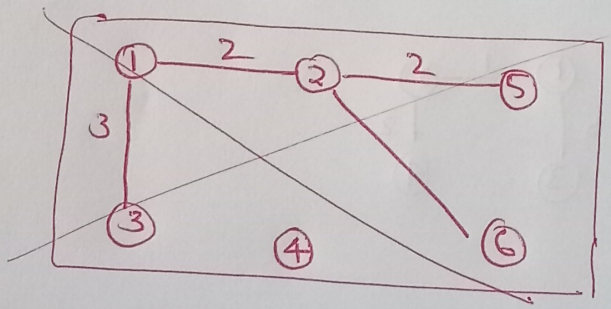
→ Select next least weight Edge i.e, (1,2) and add to Spanning Tree

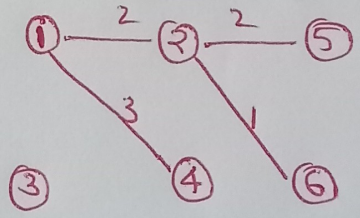


→ Next add (2,5), Edge weight = 2

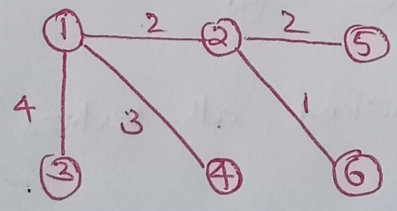


→ Next add ~~(1,3)~~ (1,4), Edge weight = 3





→ Next add (1,~~3~~4), Edge weight = 4

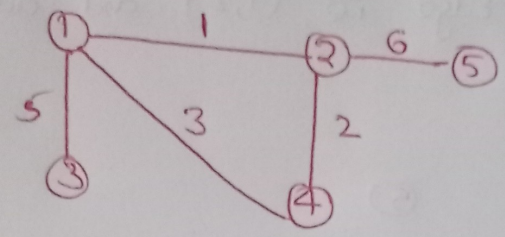


Now all vertices are covered, spanning Tree is completed

No of Edges = No of vertices - 1 = 5

Cost of spanning Tree = 12

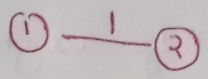
Ex :



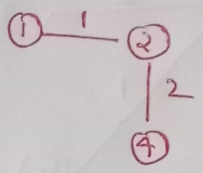
Apply Kruskal's Algorithm

Edges in Ascending order ①-②, ②-④, ①-④, ①-③, ③-⑤

Add Edge ①-②

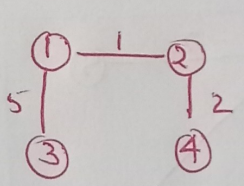


Add Edge ②-④

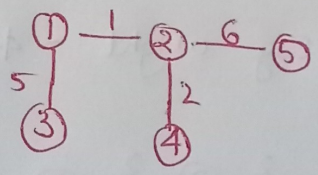


* If we add Edge ①-③ of weight 3, it forms a loop, avoid it

Add Edge ①-③



Add Edge ②-⑤



Minimal spanning is obtained all vertices are covered

cost = 14

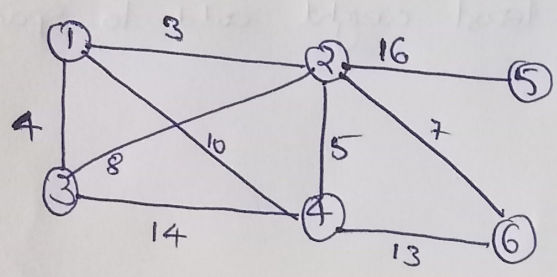
Prims Algorithm

Step 1: Select any vertex as source and start constructing tree

step 2: Find all the Edges that Connect the tree to new vertices, find minimum and it to the tree

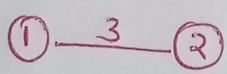
Step 3: Repeat step 2 untill all vertices are covered, such that no loop is formed

Ex:



→ Select ① as source vertex and add it to tree

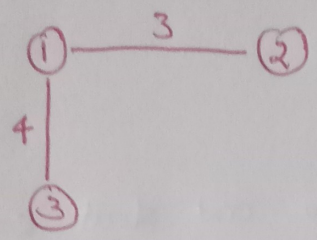
→ From ①, there are 3 Edges of weights 3, 4 & 10
Select Edge with minimum weight 3, that Exist between ① & ②



→ From ①, minimum Possible Edge is ①–③ of weight 4 ✓

②, " ②–④ " 5

among them Edge weight 4 is minimum so add add the Edge ①–④ to the spanning Tree

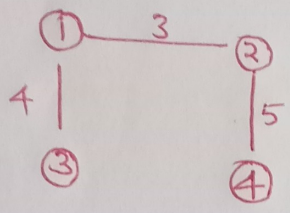


→ Now From ①, minimum Possible Edge weight is ①-④ : 8

" ③, " ③-④ : 14

" ②, " ②-④ : 5 ✓

Among them ②-④ is least weight add to Spanning Tree

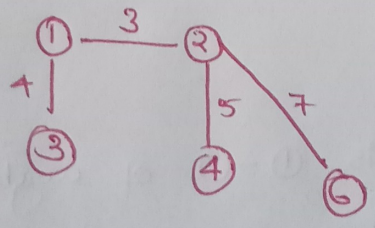


Now From ① & ③ adding any Edge forms loop so avoid them

From ②, minimum Possible is ②-⑥ of Edge weight 7 ✓

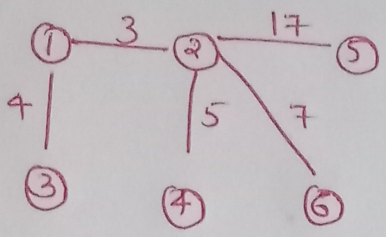
④ " ④-⑥ " 13

Add ②-⑥ to Spanning Tree



Now adding any Edge from ①, ③, ④ & ⑥ forms loop avoid.

From ②, least Possible is ②-⑤ of weight 17 ✓



✓

Finally all vertices are added to the spanning tree

Cost of spanning tree = 36

Graphs - Shortest Path Algorithms

The most commonly used shortest Path algorithms are (i) Single source shortest Path

(ii) All Pairs shortest Path

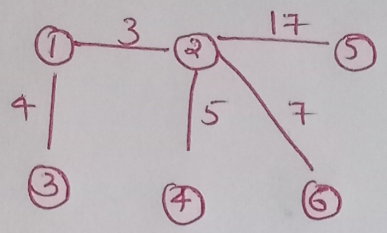
Single Source shortest Path - Any vertex from the Graph will be ~~the~~ selected as source vertex and shortest Path will be calculated to all other vertices from this source vertex

The Algorithms like Dijkstra's Algorithm & Bellman-Ford are used find the single source shortest Path

All Pairs shortest Path

In All Pairs ~~Path~~ shortest Path, the shortest distance from Each vertex to Every other vertex will be calculated

Floyd-Warshall algorithm is used to find the all Pairs shortest Path



✓

Finally all vertices are added to the spanning tree

Cost of spanning tree = 36

Graphs - Shortest Path Algorithms

The most commonly used shortest Path algorithms are
(i) Single source shortest Path
(ii) All Pairs shortest Path

Single source shortest Path - Any vertex from the Graph will be ~~the~~ selected as source vertex and shortest Path will be calculated to all other vertices from this source vertex

The Algorithms like Dijkstra's Algorithm & Bellman-Ford are used find the single source shortest Path

All Pairs shortest Path

In All Pairs ~~Path~~ shortest Path, the shortest distance from Each vertex to Every other vertex will be calculated

Floyd-Warshall algorithm is used to find the all Pairs shortest Path

Single Source Shortest Path Algorithm – Dijkstra's Algorithm

Algorithm

Step 1 : Select the source vertex and mark it as visited with current distance as 0 (zero). For remaining vertices Set distance to infinity

Step 2 : For each neighbouring vertex v of current vertex u, add the distance of u i.e, d[u] to the weight of the Edge connecting u-v i.e., C(u,v)

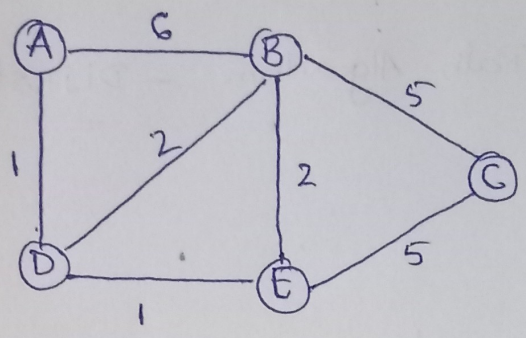
If their sum is smaller than current distance of v i.e., d[v], set it as d[v]

$$d[v] = \min \{ d[v], d[u] + C(u,v) \}$$

Step 3 : Mark the current vertex ~~as visited~~ with minimum distance as visited. For all it unvisited neighbouring vertices apply step 2

Step 4 : Terminate the Process, when all vertices are visited

Ex :-



→ Select vertex A as source vertex, update its distance as zero and remaining vertices distance as ∝

A	0
B	∝
C	∝
D	∝
E	∝

Mark A as visited

visited = { A }

unvisited = { B, C, D, E }

→ If A have any neighboring vertices update their distances from A

A	0
B	6
C	∝
D	1
E	∝

visited = { A }

unvisited = { B, C, D, E }

→ From A, direct edge Exists with unvisited vertices B and D only

→ From this updated distance table, Select vertex with minimum distance which is 'D'

→ If D have any unvisited neighbouring vertices update their distances from D

$d[v] \rightarrow$ | A | 0 |
|---|---|
| B | 6 |
| C | ∞ |
| D | 1 |
| E | ∞ |

$V = \{A, B, C, D, E\}$

Visited = {A}

unvisited = {B, C, E}

u = D

B, $d[B] = 6$,

$d[D] + c(D, B) = 1 + 2 = 3$

Existing distance to B can be reduced if we can reach through vertex 'D'

$\therefore d[B] = d[D] + c(D, B) = 3$ ✓

C, Here C is not a neighboring vertex to D, so we can not minimize Existance distance

E, $d[E] = \infty$

$d[D] + c(D, E) = 1 + 1 = 2$

Existing distance to E can be reduced, it we can reach through vertex 'D'

$\therefore d[E] = 2$ ✓

Now updated d[v] will be, Next minimal distance 'E'

A	0
B	3
C	∝
D	1
E	2

visited = { A, D }

unvisited = { B, C }

u = E

B, d[B] = 3

d[E] + c(E, B) = 2 + 2 = 4

Existing distance to B is lesser compared to distance through vertex E, so do not update

C, d[C] = ∝

d[E] + c(E, C) = 2 + 5 = 7

Existing distance is ∝, but we can reach vertex C with distance if reach through E so update distance[C] as 7

A	0
B	3
C	7
D	1
E	2

→ Now select minimum distance from unvisited vertices i.e., vertex B

visited = { A, D, E }

unvisited = { C }

u = B

C, d[C] = 7

d[B] + C(B,C) => 3 + 5 = 8

distance to C through B is not minimized

do not update the Existing distance

A	0
B	3
C	7
D	1
E	2

→ Now select last unvisited vertex C, From which we can not reach any unvisited vertex, No updation in distances

visited = { A, B, C, D, E }

unvisited = { }

All vertices are visited, Algorithm Ends with Final distance as

A	0
B	3
C	7
D	1
E	2

=> shortest distances from Source vertex 'A' to all other vertices

All-Pair shortest Path - Floyd-Warshall algorithm

Step 1: Store the graph in adjacency matrix, where d(i,j) represents distance between vertex i & j

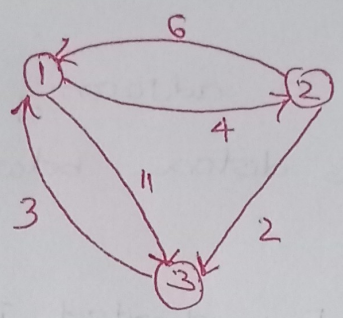
Step 2: select a vertex, Find shortest Path to all other vertices from this vertex & update adjacency matrix

Step 3: Repeat step 2 for all other vertices

Step 4: The Final adjacency matrix, contains all Pairs shortest Path

```
for k = 1 to n                    // n -> no of vertices
{
    for i = 1 to n
    {
        for j = 1 to n
        {
            d(i,j) = min{d(i,j), d(i,k)+d(k,j)}
        }
    }
}
```

Find All Pairs shortest Path for following graph



For Given graph, the initial cost adjacency matrix

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{array}$$

All Pairs shortest Path

$$= \min \left\{ \{ A^{k-1}(i,k) + A^{k-1}(k,j) \}, C(i,j) \right\}$$

Here No of vertices = 4

$\Rightarrow$ k runs from 1 to 4

$\rightarrow$ Set k = 1, Select any vertex say ① ~~first~~ find shortest Path between all Pairs of vertices with ① as intermediate vertex

$$A^1 = \min \left\{ \{ A^0(i,1) + A^0(1,j) \}, C(i,j) \right\}$$

$A^1(0,0)$ $A^1(1,1) = 0$, $A^1(2,2) = 0$, $A^1(3,3) = 0$

$A^1(1,2) = 4$

$A^1(1,3) = 11$

} No change in distance because 1 is source as well as the intermediate vertex

$A^1(2,1) = 6$

$A^1(3,1) = 3$

} No change in distance because 1 is intermediate as well as the End vertex

$A^1(2,3)$

$= \min \{ \{A^0(2,3)\}, \{A^0(2,1) + A^0(1,3)\} \}$

$C(2,3)$

$= \min \{ 2, (6+11) \}$

$= 2$ // No need to update distance

$A^1(3,2) = \min \{ \{A^0(3,2)\}, \{A^0(3,1) + A^0(1,2)\} \}$

$= \min \{ \infty, (3+4) \}$

$= 7$ // Need to update

Now

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

Now Set K = 2, Select any vertex say ② find shortest Path to other vertices with ② as intemediate

$$A^2 = \min\{\{A'(i,2)+A'(2,j)\}, C(i,j)\}$$

$$A^2(1,1) = A^2(2,2) = A^2(3,3) = 0$$

$A^2(2,1) = 6$
$A^2(2,3) = 2$
} ② is Source / intemediate
No change in distance

$A^2(1,2) = 4$
$A^2(3,2) = 7$
} ② is intemediete / End
No change in distance

$A^2(1,3)$

$= \min\{\{A'(1,2)+A'(2,3)\}, C(1,3)\}$

$= \min\{(4+2), 11\}$

$= 6$ // update

$A^2(3,1)$ $= \min\{\{A'(3,2)+A'(2,1)\}, C(3,1)\}$

$= \min\{(7+6), 3\}$

$= 3$ // No need to update

Now

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

Now set $k=3$, Select the last vertex ③ and find Shortest Path to all vertices with ③ as intermediate

the I_3^3 =

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

We have calculated shortest Path from ~~Each~~ Each vertex to Every other vertex and $k3$ is the cost matrix with All Pairs shortest Path