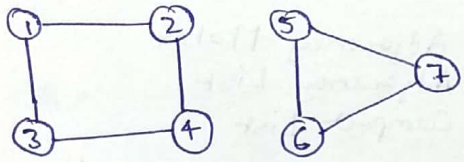
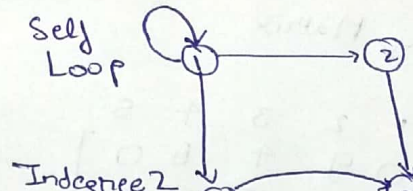
Graphs

Non-Connected



2 components

Self Loop

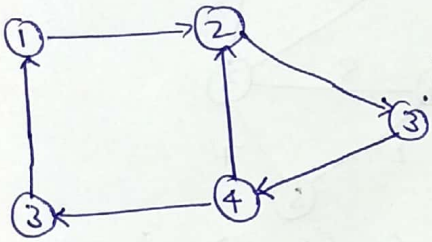


Indegree 2
outdegree 1

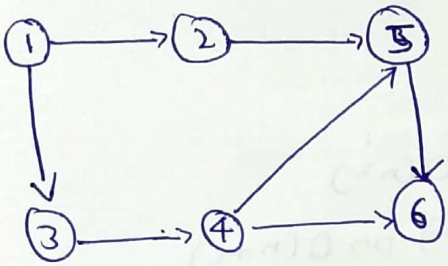
Parallel Edges

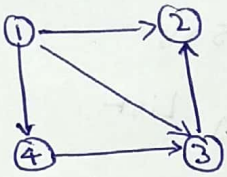
Directly Graph

Strongly-Connected



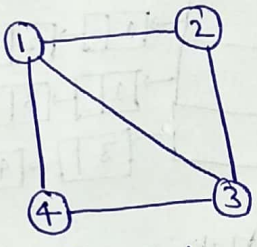
Directed Acyclic Graph
(DAG)





Simple Digraph
[Directed graph]

Degree 3



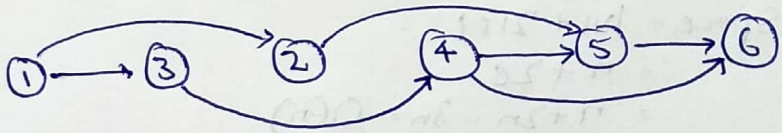
Graph /
Non-Directed Graph

Articulation Point → Removing that vertice from the graph, the graph will become non-connected graph.

Strongly Connected Graph → If you can reach any vertex from any other vertex.
A path exists b/w any two pair of vertices

Path → Set of all the vertices in between a pair of vertex.

Cyclic Path → Starts from one vertex and ends at the same vertex.
or "Cycle"



Topological ordering
DAG can be arranged linearly such that edges going in forward direction

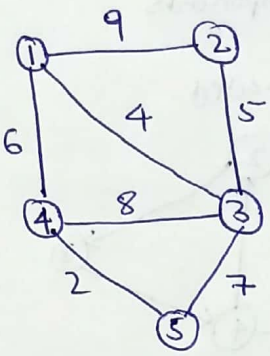
Representation of Undirected Graph

Adjacency Matrix

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 9 & 4 & 6 & 0 \\ 9 & 0 & 5 & 0 & 0 \\ 4 & 5 & 0 & 8 & 7 \\ 6 & 0 & 8 & 0 & 2 \\ 0 & 0 & 7 & 2 & 0 \end{bmatrix} \end{array}$$

5×5

1. Adjacency Matrix
2. Adjacency List
3. Compact List

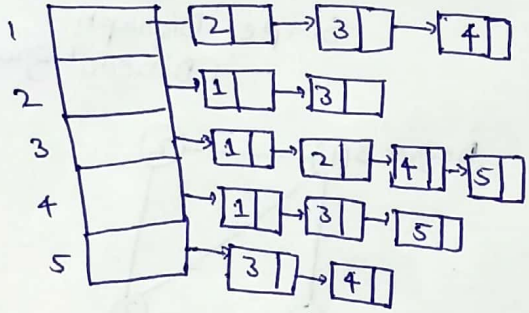


$G = (V, E)$

$|V| = n = 5$

$|E| = e = 7$

Adjacency List



Adjacency Matrix → $n \times n = n^2$ TC = $O(n^2)$

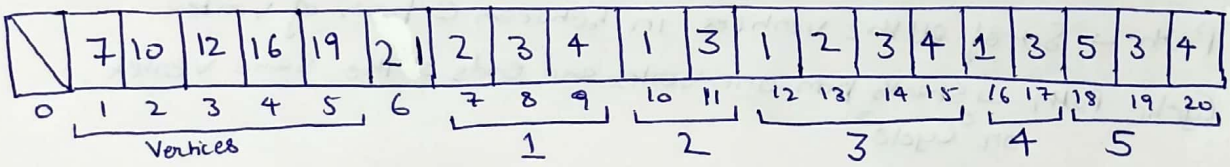
Adjacency List → $|V| + 2|E| = O(n + 2e)$ or $O(n+e)$

→ Similarly in Adjacency List, we should store weight in each node as well.

Compact List Representation

→ Single dimension Array

Size of Array = $|V| + 2|E| + 1$



Space = $|V| + 2|E|$

$= n + 2e$

$= n + 2n = 3n = O(n)$

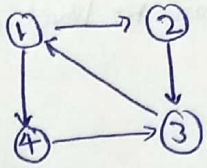
Representation of Directed Graphs

$A =$

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

4x4

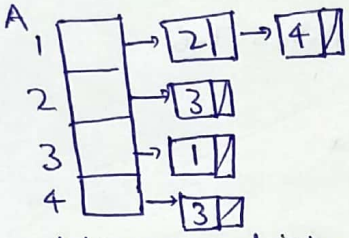
Adjacency Matrix



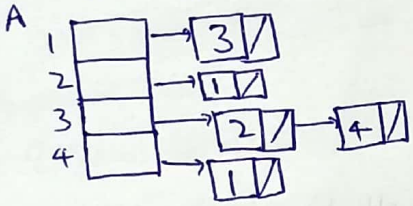
G(V, E)

|V| = 4 |E| = 5

Indegree → Check Column
Outdegree → Check Row



Adjacency List → (Outdegree)



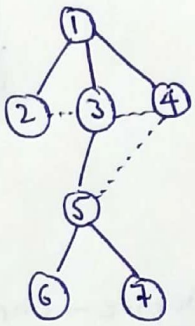
Inverse Adjacency List → (Indegree)

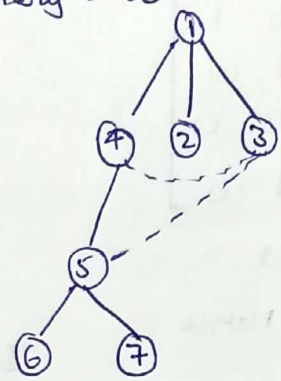
Time taken = $O(n^2)$
if Adjacency Matrix used

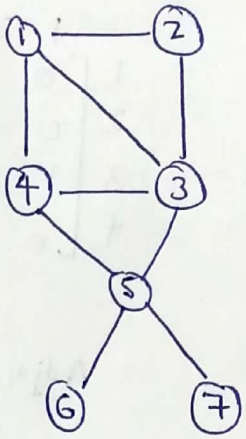
if Adjacency List used,
Time taken = $O(n)$

Breadth First Search (BFS)

→ Similar to level order of Binary Tree

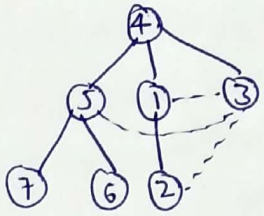






BFS: 1, 2, 3, 4, 5, 6, 7 BFS: 1, 4, 2, 3, 5, 7, 6

→ For 1 graph, multiple BFS possible.
→ We can start in any order, with any vertex

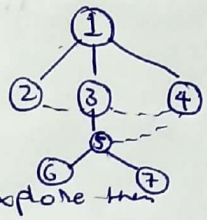


BFS: 4, 5, 1, 3, 7, 6, 2

BFS: Start from any vertex you like.
→ When you visit a vertex, explore all it's adjacent vertices
→ While exploring, you can visit adjacent vertex in any order

1) Visiting 2) Exploring

BFS: 1, 2, 3, 4, 5, 6, 7



Queue: ~~1, 2, 3, 4, 5, 6, 7~~

→ Select vertices from queue, one by one and explore them
→ Once you visit a vertex, add it to the queue.

BFS Spanning Tree.
Dotted Edges → Cross Edges
Time = O(n)

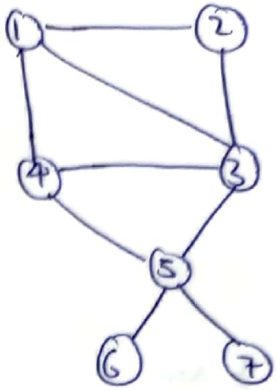
eg. 1) 2, 3, 1, 5, 4, 7, 6
2) 5, 7, 3, 6, 4, 2, 1
3) 1, 4, 3, 2, 5, 7, 6
] all correct

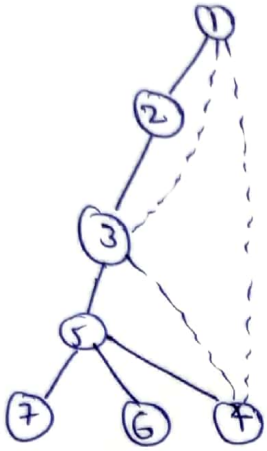
For program: Maintain
i) Queue → Enqueue & Dequeue → Insert & Delete
ii) Adjacency Matrix
iii) Visited Array [Flag for visited vertex]

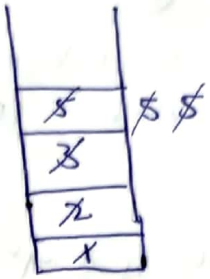
Depth First Search (DFS)

→ Similar to pre-order traversal

1. Visited 2. Exploring







Uses a Stack

→ When you get a new vertex, suspend previous vertex

→ Explore the new vertex.

→ You can start from any vertex.

→ DFS Spanning Tree

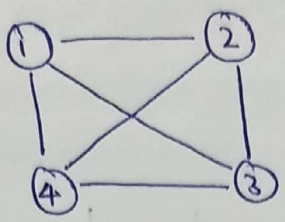
Back Edge → Connecting to previous level Node

Time = O(n) [Analytical Time]

eg. 1) 1, 3, 5, 4, 7, 6, 2
2) 1, 2, 3, 4, 5, 6, 7
3) 1, 4, 5, 7, 3, 2, 6

] all Correct

Spanning Tree



1. Spanning Tree
2. Prims MST
3. Kruskals MST

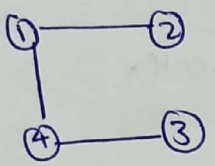
$G = (V, E)$

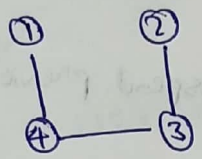
$|V| = n \quad |E| = e$

$S \subseteq G$

We can find 'S' only if graph is connected.

$S = (V', E') \quad |V'| = |V| \quad |E'| = |V| - 1$





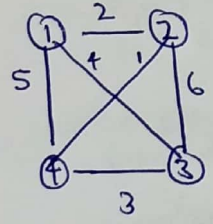
→ More than one spanning Tree for same graph.

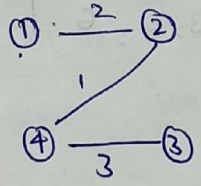
→ 'S' Conditions: i - Connected ii - No Cycle

${}^{|E|}C_{|V|-1}$ ${}^{6}C_{3}$ → Total possible the reject with Cycles

${}^{6}C_{3} - 4 = 20 - 4 = 16$ Spanning Trees

Minimum Cost Spanning Tree





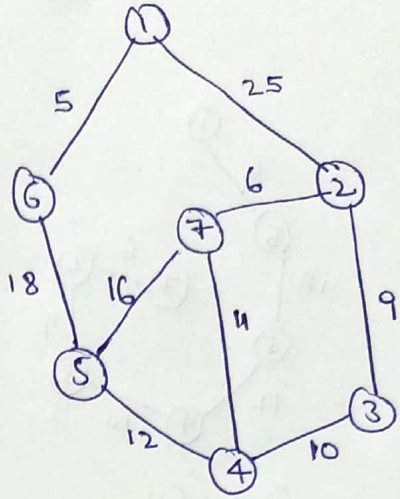
They all select best with minimum cost.

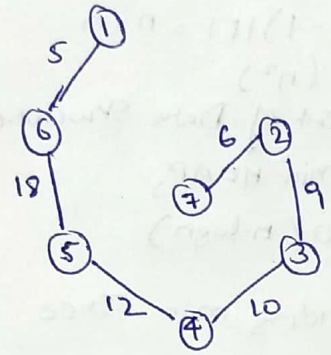
Prim's Algorithm

→ Select minimum cost edge from graph.

Now, Select next minimum cost edge but it should be connected to the previous edge.

Repeat.





Total Cost = 5 + 18 + 12 + 10 + 9 + 6 = 60

Time = (|V| − 1)|E|

= n e × n × n ≈ O(n²)

If we use heap we can reduce time : (|V|−1) log(E|

= O(n log n)

For writing program:

i) Identify the type of Data Structure required.

Graph represented by Adjacency Matrix.

ii) Take 2-D array to store the Spanning Tree.

iii) 'nearest' Array → (size = (|V|+1)] Used to check adjacent.

Check either lower Triangular / Upper Triangular & find min. weight. Write down in the 2-D array.

→ In 'nearest' array write down which vertex is close to which of the adjacent vertex previously selected in Spanning Tree.

→ Now, Repeating steps:

From 'nearest' find out minimum.

Write that edge in Spanning Tree Result Array.

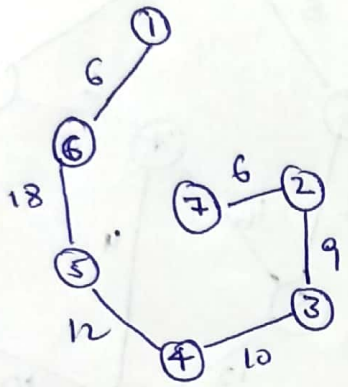
Update 'nearest' array.

Mark the included vertex as '0'.

Repeat.

Kruskal's Algorithm

Spanning Tree



Cost = 60

1) Always select the min. weight from the graph
2) Make sure that vertex is not forming a cycle.

Time = $(|V|-1)|E| = n \times n$
$= O(n^2)$
(Independent of Data Structure)

If we use min HEAP,
$O(n \log n)$

In Kruskal's → more focus on finding min. Tree

Prim's → more focus on finding Tree

Kruskal's Algorithm – If 2 non-connected graphs
Finds spanning tree for each graph individually.

Prim's Algorithm – Find spanning Tree for only one of the individual graph.

Steps:

1) Find out min. cost edge
2) Find out if it is forming a cycle or not
3) Perform weighted union upon them.

] Repeat

→ If diff. parents we can join new vertices, as it implies it's not forming a cycle.

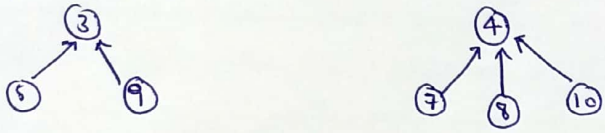
Disjoint Sets

$U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

S

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8	9	10

$A = \{3, 5, 9\}$ $B = \{4, 7, 8, 10\}$ $A \cap B = \phi$



-1	-1	-1	-3	-4	3	-1	4	4	3	4
0	1	2	3	4	5	6	7	8	9	10

Disjoint sets → If we take intersection, no common elements

-ve no. shows it's a set.

So in A, 5 and 9 are children of 3.

At 3, we write -ve → to show it's a Head.

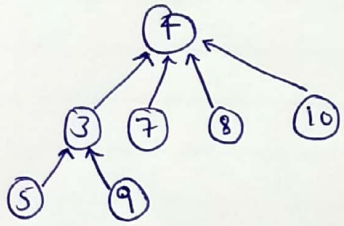
'magnitude' shows the total no. of nodes.

1) Union or Weighted Union

Perform $A \cup B = \{3, 4, 5, 7, 8, 9, 10\}$

→ Whichever has more elements select that one as parent.

So, in this case → 4



S

	-1	-1	4	-7	3	-1	4	4	3	4
0	1	2	3	4	5	6	7	8	9	10

2) Finding or Collapsing Find

→ To find the parent

→ Check for negative numbers in the array.

→ We just have to follow hierarchy and reach the root.

If 2 vertices from different set connect then.

If " " same set don't connect then

else they will form a cycle.

Kruskal's Program:

1) Take Array of Edges (No. of Columns = No. of Edges)

Now Rows, 2 → vertices, 1 → weight. Total = 3 Rows

2) Set List → For detecting a cycle. Size = No of Vertices + 1

3) Included List → To 'flag' if edge is included or not

4) Final Solution 2-D Array → Data Structure