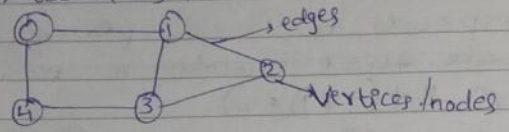
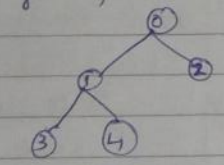
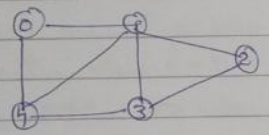
06 = Graphs

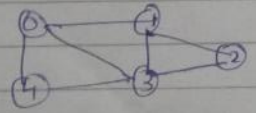
* Graph is a non-linear d.s consisting of node & edges.
* Nodes → vertices.
* eg →



* Tree vs graph =
Every tree is a graph, but every graph is not a tree.

Tree	Graph
* Source to destination is single path	* multiple path
	
tree	Graph
ഇതിന്റെ 0-ൽ നിന്ന് 3-ലേക്ക് പോകാനായിട്ട് 0-1-3 only possible.	Here 0-ൽ നിന്ന് 3-ലേക്ക് പോകാനായിട്ട് multiple path. ഉണ്ട് → 0-1-3, 0-4-3 ...
* root node will be there.	* All nodes are equal.

→ Mathematical repren of graphs =



$G = (V, E)$

G = graph
V = set of vertices
E = set of edges

$V(G) = \{0, 1, 2, 3, 4\}$.
$E(G) = \{(0,1), (0,4), (0,3), (1,3), (1,2), (2,3)\}$.

$G = (V, E)$ → are parentheses to indicate unordered pair
$G = \langle V, E \rangle$ → angled brackets to indicate ordered pair.

→ Application of graph =
* Google maps & GPS
  → To find shortest path.
* Facebook → To represent user connections.

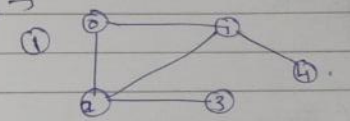
⇒ Types of graphs =

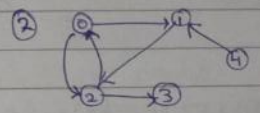
* Based on (dir) of edges → (2 types)

a) Undirected (g) ⇒ A graph in which all the edges are bi-dir.

b) Directed (g) ⇒ A graph in which all the edges are uni dir.

eg →





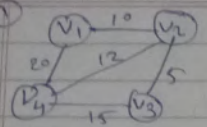
undirected (g)
ഇതിന്റെ നമുക്ക് 0-1 ഉം 1-0 രണ്ടായും പോകാം,
bcz specified (dir) ഒന്നും ഇല്ല.

directed (g)
(dir) specify ചെയ്തിട്ടുണ്ട്.

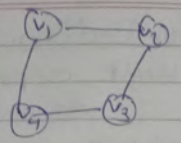
* Based on weight of edges → (2 types)

a) weighted (g) = A (g) in which each edge is assigned with some weight / cost / value.

b) unweighted (g) = A (g) where there is no value / weight associated with edges.



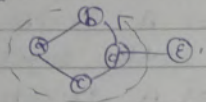
weighted (g)



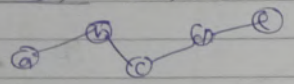
unweighted (g)

* Based on cycles in (g) — (2 types)

a) cyclic (g) → atleast 1 cycle will be there



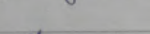
b) acyclic (g) → no cycle will be there.



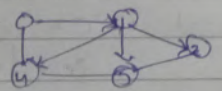
=> graph terminologies =

1) Adjacent nodes / neighbour nodes =

Node x is adjacent node y if there is an edge from node x to node y.

eg→ ① 

o is adjacent to 1 & vice versa.
bcz it is bi-dirn (g).

②. 

o is adjacent to 1 but 1 is adjacent from o bcz it is (dirg) (g).

2) path = sequences of vertices in which each pair of successive nodes is connected by an edge.

eg→

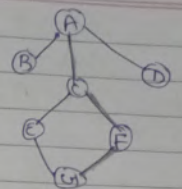


fig: 1.1

A − G =>
A − C − F − G or
A − C − E − G

Length of a path = no. of edges in that path
here length of A−G = 3 (3 edges).
" of A−F = 2

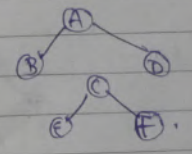
Length of path >= 1

2 types of path = ① Simple path = vertices are distinct.
eg→ A C F G → in fig 1.1

② closed path = 1st & last node of that path is same.
eg→ C F G E C.

③ cycle = starting & ending nodes are same, all other nodes are distinct
eg→ C F G E C.

3) connected (g) = if there is a path from any node to any other node.
eg→ fig: 1.1
✓
B−C ✓
A−G ✓



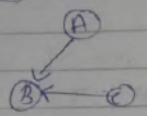
X.
A−C X

2 types →

1) Strongly c. (g) =

A → C ✓
A → B ✓
C → A ✓

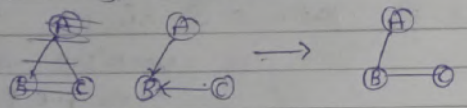
② weakly C.(g) =



A — C X.
C — A X.

* not a strongly C.(g).
* But if we are converting this to undirected (g) all nodes are connected.

eg →



A — C = ABC ✓
C — A = CBA ✓

that's why it is weakly C.(g)

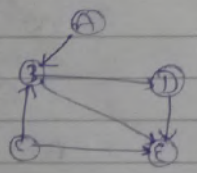
✓ ④ Degree = No. of edges connected to a node.

2 types →

a) Indegree of a node = no. of edges coming to that node.

b) outdegree of a node = no. of edges going outside from that node

eg →

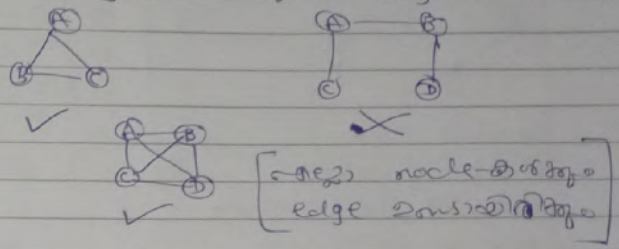


Indegree (D) = 1
outdegree (D) = 1
Indegree (E) = 3
outdegree (E) = 0

⑤ complete graph =

It is a simple undirected (g) in which every pair of distinct vertices is connected by a unique edge.

eg →



[eg) node-B of (g) edge connected (g)]

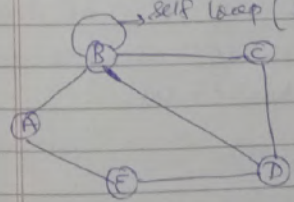
⇒ Representation of graph =

① Adjacency matrix =

* It is a sequential repre.
* used to represent which nodes are adjacent to each other. (i.e) there any edge connecting nodes to a graph.
* In this repre. we have to construct a n×n matrix A.
* If there is any weighted graph then instead of 1s and 0s, we can store weight of the edge.

eg →

① Undirected (g) →



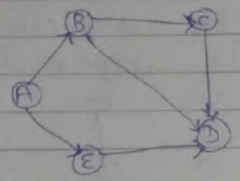
⇒

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

* consider the rows.
→ A ⇒ connected to B & E
so 1, others 0

Adjacency Matrix → Directed (g)
→ Undirected (g)
→ weighted (g)

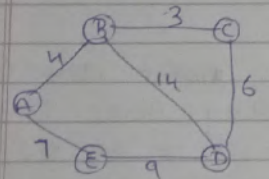
② Di graph (g) →



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

consider the outgoing nodes. take it as 1, others all 0.

③ Undirected weighted (g) →



	A	B	C	D	E
A	0	4	0	0	7
B	4	0	3	14	0
C	0	3	0	6	0
D	0	14	6	0	9
E	7	0	0	9	0

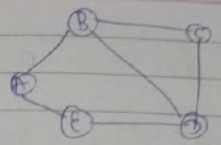
consider the weight instead of 1 & 0.

② Adjacency list =

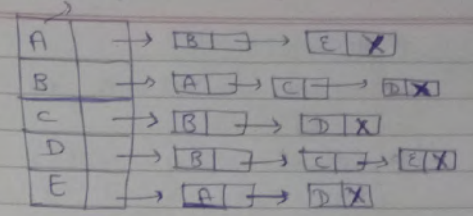
* It is a linked repre~
* In this repre~, for each vertex in (g), we maintain the list of its neighbours. It means, every vertex of (g) contains list of its adjacent vertices.

eg →

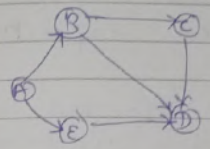
① Undirected (g) →

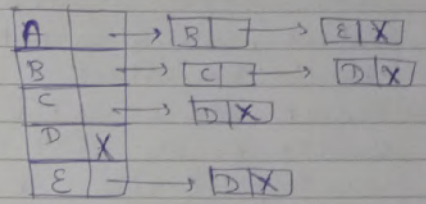


A is connected to B & E.



② Directed (g) →





⇒ Graph traversal =

* It is a technique used for searching a vertex in (g).
* Also used to decide the order of vertices in the search process.
* 2 types →
  a) DFS (Depth First search) → (stack)
  b) BFS (Breadth First search) → (Queue)

a) DFS =

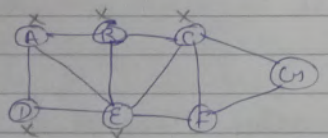
* It produces spanning tree as final result if (g) a graph without loops.
* We use stack ds with max size of total no of vertices in (g) to implement DFS traversal of a (g).

Stack → LIFO
queue → FIFO

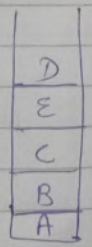
* Algo →

1) Define a stack of size total no. of vertices in (G).
2) Select any vertex as starting point for traversal, visit that vertex & push it on to the stack.
3) visit any 1 of the adjacent vertex of the vertex which is at top of the stack, which is not visited, & push it on to the stack.
4) Repeat stp 3 until there are no new vertex to be visit frm the vertex on top of the stack.
5) when there is no new vertex to be visit then use backtracking & pop 1 vertex frm stack.
6) Repeat stp 3, 4, 5 until stack becomes empty.
7) when stack becomes empty, then produce final spanning tree by removing unused edges frm the graph.

eg →

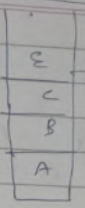


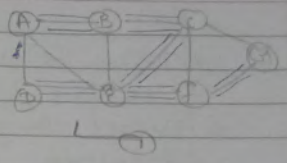
* Select any starting node → A.
* adjacent node of A → B, D, E → B.
* then push to stack.
* B → C
* C → E
* E → D
* D → A X So backtracking (B) & pop top elemnt.



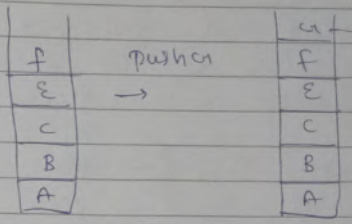
Stack.

stack empty → first element add → pop → last stack empty → we visited all nodes

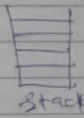




→ visit F.

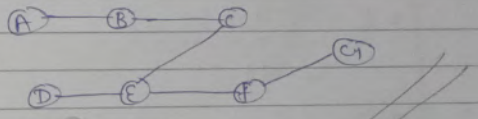


* G1 → No visited node So (B) & pop.
→ G1 X
* Check f " " So (B) & pop.
→ F X
* until A. , So (B) & pop
* Stack become empty.



Stack

Spanning tree



weight

A B C E D F G1 visited

Stack.

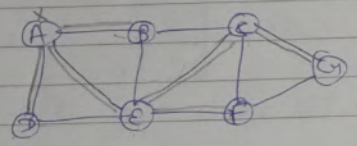
b) BFS =

* BFS produces a spanning tree as a final result i.e. graph without loops.
* we use queue d.s with max size of total no. of vertices in graph to implement BFS of a (G).

* (A1) =

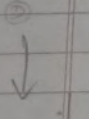
1) Define a queue of size total no of vertices in G.
2) Select any vertex as starting point of traversal, visit that vertex & ins it onto the q.
3) visit all adjacent vertices of vertex which is at point of q which is not visited & ins them into the q.

eg → ①

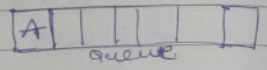


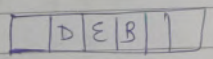
→ visited edges only

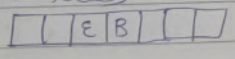
visit all vertex of A.

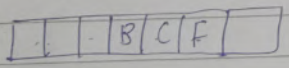


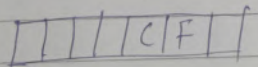
* Select any vertex → A.
* adj vertexes of A → D, E, B
  ins them & dlt A. frm Q.
* Select front vertex → D
* adj vertexes of D → A & E (A visited)
  & dlt D.
* E adj A, D, B, C, F
  & dlt E.
* B → A, E, C & dlt B
* C → B, E, G, F & dlt C
* F → G & dlt F
* G → no vertex.
  So dlt G

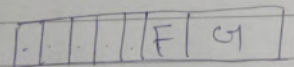


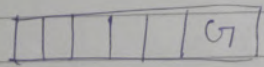


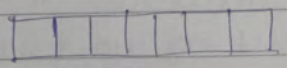












Spanning tree of BFS →