Graph

A Non linear data structure that consists of a finite set of vertices or nodes and a set of edges connecting them

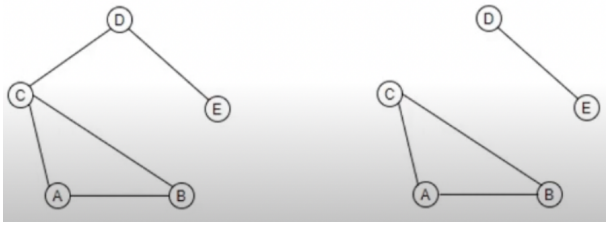
Vertices: where data is stored
Edges: which connects the vertices

Applications of graph:

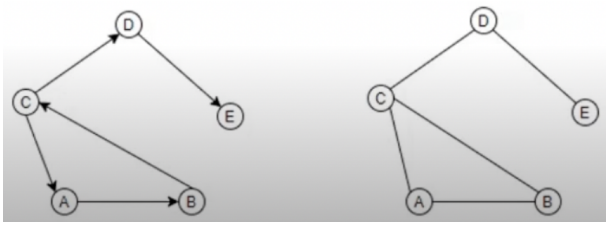
1. Facebook: users are nodes. So like if two nodes are directly connected then users are friends. If they are indirectly connected then users are mutual friends.
2. WWW: Each page is a node and edges are basically like hyperlinks which connects 2 pages
3. Google Maps: Tries to find shortest time to reach destination
4. Flight Network: Tries to find the best route with minimum fuel and minimum cost
5. Product Recommendations: If user A buys two products they are connected and stored. Then if user B buys one of those products, amazon will recommend the 2nd product to user B.

Terminologies:

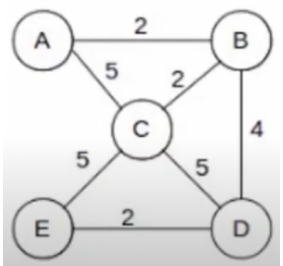
1. Connected/Disconnected graph: A graph is said to be connected if there exists at least one path between every pair of vertices. Otherwise it is disconnected.



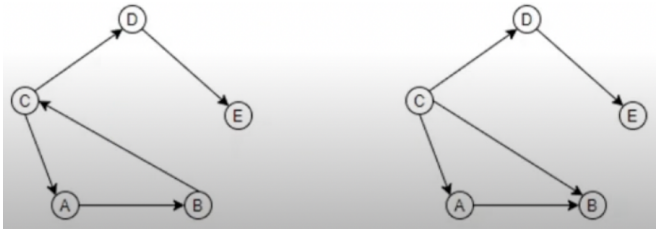
2. Directed/Undirected graph: In directed graph, each edge has a direction which determines the traversal order. In undirected graphs, the edges are unidirectional so can be traversed in a graph in either ways.



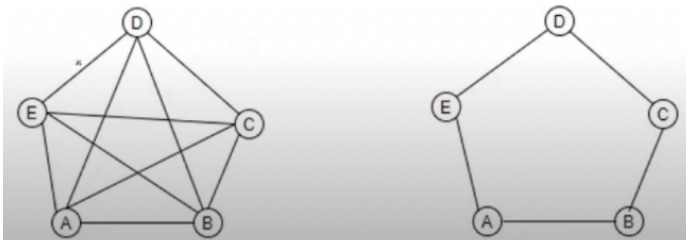
3. Weighted graph: A graph where each edge has a numerical weight assigned to it.



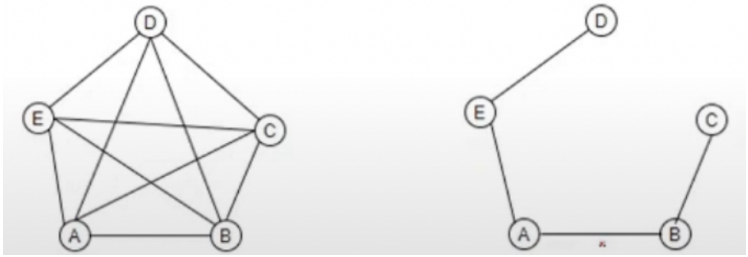
4. Cyclic/Acyclic graph: A directed graph containing at least one graph cycle. A graph having only one cycle is called unicyclic graph. Acyclic graph(DAG-Directed Acyclic Graph) does not have cycle



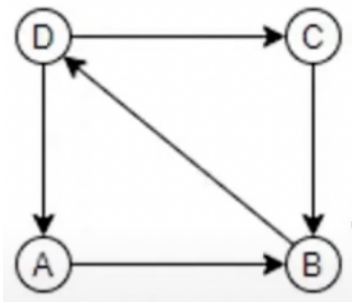
5. Dense/Sparse graph: A graph in which the number of edges is close to the maximal number of edges. Whereas in sparse graph the number of edges is close to the minimal number of edges.



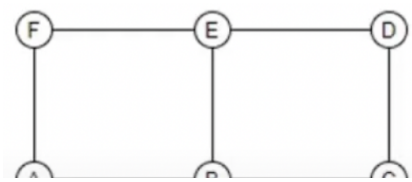
6. Simple/Complete graph: Simple graph has no loops and no multiple edges between two nodes. A complete graph is a undirected graph in which every pair of distict vertices is connected by a unique edge. Every complete graph is a simple graph.



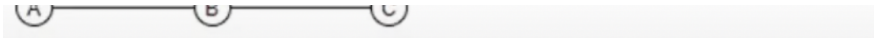
7. Strongly Connected graph: A directed graph is strongly connected if there is a path in each direction between each pair of vertices of the graph. Like D->A and A->B->D. So A and D have a path in both directions.



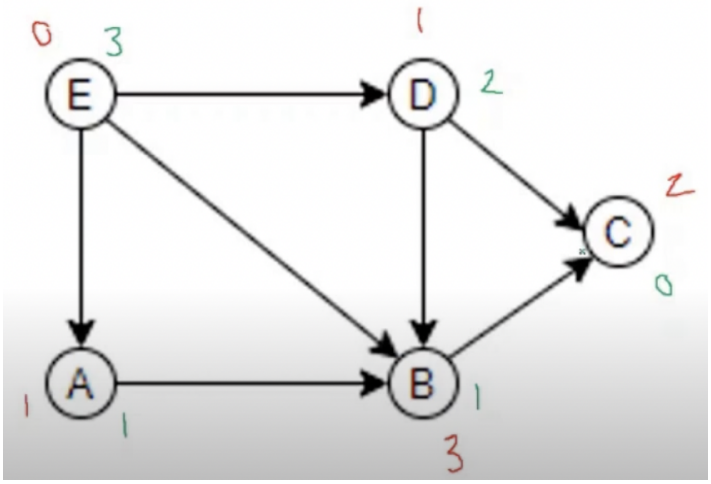
8. Adjacency: Adjacent vertices have an edge between the vertices and Adjacent edges have a common vertex between them



Adjacent vertices - <A,B>, <A,F>
Adjacent edges - <AB,BE>, <AF,FE>



9. Degree: For undirected graph, it is number of vertices adjacent to vertex V.
   In directed graph:
   1. Indegree: Number of incoming edges (Red)
   2. Outdegree: Number of outgoing edges (Green)



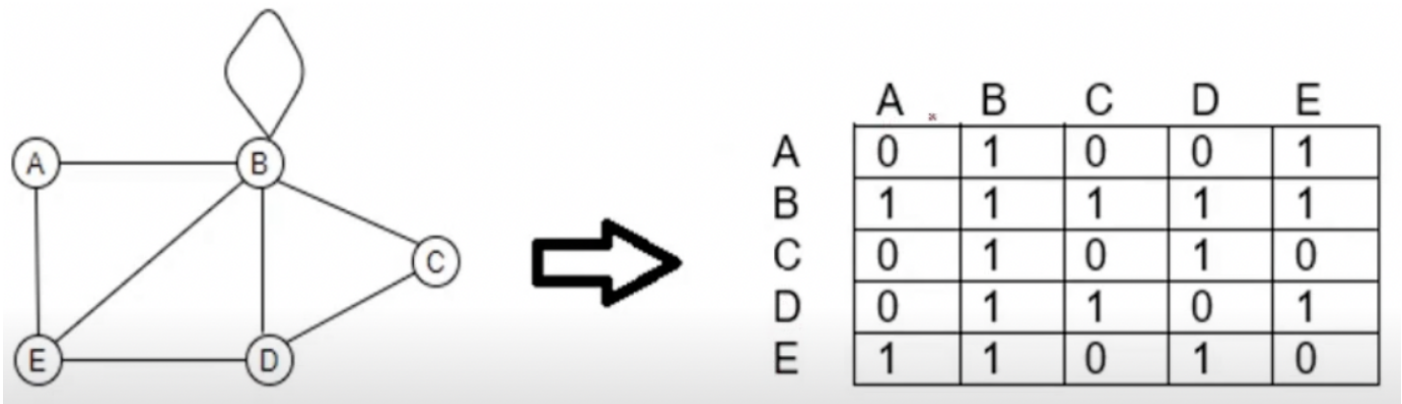
Sum of indegree and outdegree is always even. Why? Because each edge is counted twice, one for outgoing and one for ingoing.

10. Path: Sequence of distinct vertices such that two consecutive vertices are adjacent.
11. Cycle: A closed path is a cycle i.e a path in which the only repeated vertices are the first and last one
12. Walk: Sequence of vertices and edges i.e. if we traverse a graph in any way then we get a walk. Vertices and edges can be repeated.

Representation of Graph:

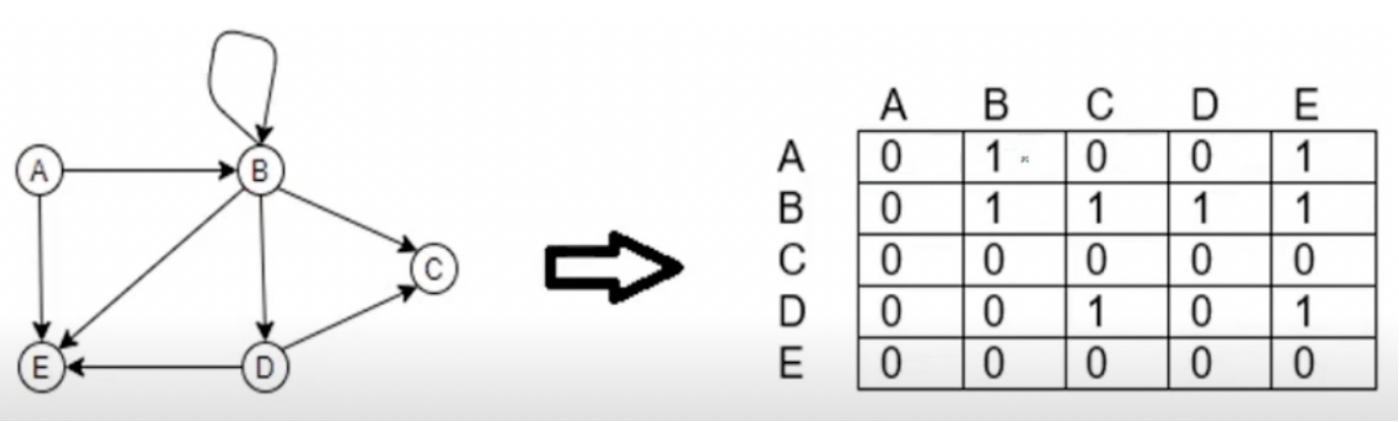
1. Adjacency matrix
2. Adjacency List
3. Incidence Matrix

Adjacency Matrix: It is a 2D array of size V X V where V is the number of vertices in a graph. The matrix is filled with 0's (edge not present) and 1's (edge present) if it is not weighted. If it is weighted graph, 1 will be replaced by the weight of the edge.



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

For directed graph check the direction of edge also.



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

Advantages:

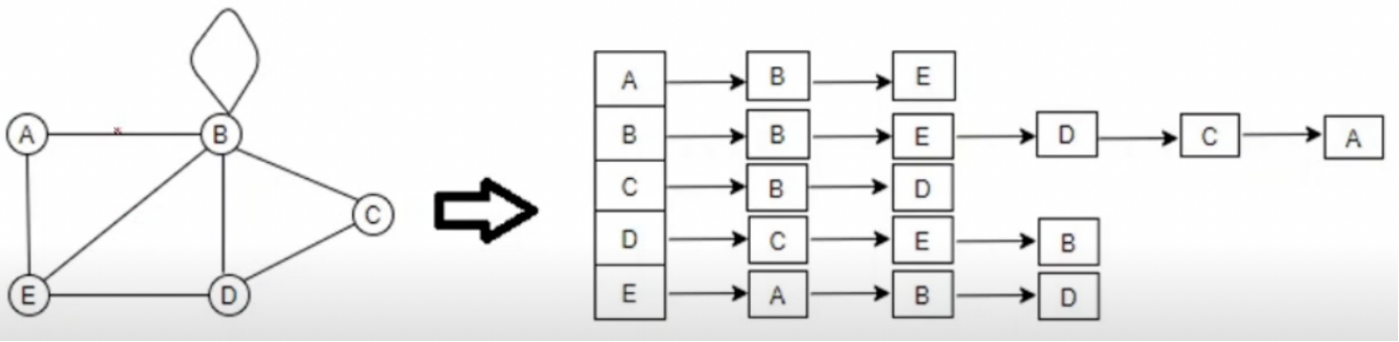
1. Easy to understand and implement
2. Adding and removing an takes O(1) time because you just need to go to the correct position and replace the value
3. Queries like whether there is an edge from vertex u to v is also very efficient and can be done in O(1) time.

Disadvantages:

1. Consumes a lot of space. And like even if the graph is sparse i.e. really less edges still the size of matrix will be V X V
2. Adding or removing a vertex is O(V^2) time

```
#include<bits/stdc++.h>
using namespace std;
#define V 5

void addEdge(int mat[][V],int u, int v){
    mat[u][v]=1;
}

int main(){
    int mat[V][V]={0};
    addEdge(mat,0,1);
    addEdge(mat,0,4);
    addEdge(mat,1,1);
    addEdge(mat,1,2);
    addEdge(mat,1,3);
    addEdge(mat,1,4);
    addEdge(mat,3,2);
    addEdge(mat,3,4);
    for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            cout<<mat[i][j]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

Adjacency List: Collection of unordered lists used to represent a finite graph. Each list describes the set of neighbours of a vertex in the graph.



If it is a weighted graph elements in the list will contains pairs like B,2 i.e. node, weight

Advantages:

1. Saves space O(|V|+|E|)
2. Adding an edge takes O(1) time. (Just go to the required list and add a new element at the end)
3. Adding a vertex takes O(1) time. (Just add another element in array)

Disadvantages:

1. Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V)
2. Time taken to remove an edge takes O(E) time
3. Time taken to remove a vertex O(V+E) time

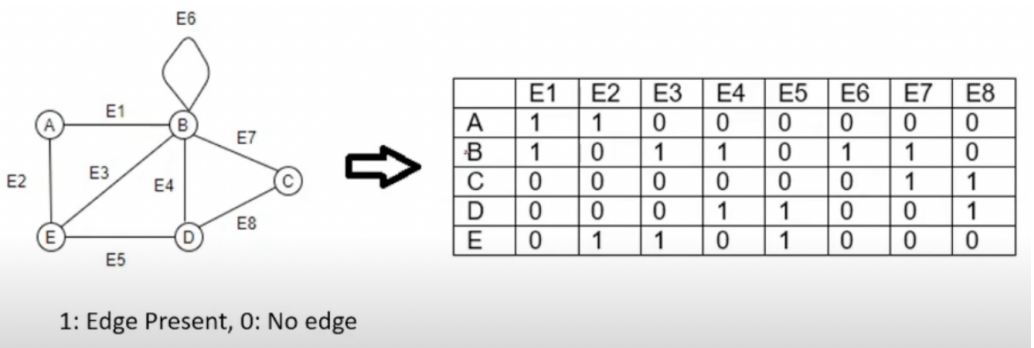
```
#include<bits/stdc++.h>
using namespace std;
#define V 5

void addEdge(vector<int> graph[V],int u, int v){
    graph[u].push_back(v);
}

int main(){
    vector<int> graph[V];
    addEdge(graph,0,1);
    addEdge(graph,0,4);
    addEdge(graph,1,1);
    addEdge(graph,1,2);
    addEdge(graph,1,3);
    addEdge(graph,1,4);
    addEdge(graph,3,2);
    addEdge(graph,3,4);
    for(int i=0;i<V;i++){
        cout<<i;
        for(int j=0;j<graph[i].size();j++){
            cout<<"->"<<graph[i][j];
        }
        cout<<endl;
    }
    return 0;
}
```

For weighted graph:

```
#include <bits/stdc++.h>
using namespace std;
#define V 5

void addEdge(vector < pair<int, int> > graph[V], int u, int v, int weight)
{
    graph[u].push_back(make_pair(v, weight));
}

int main()
{
    vector<pair<int,int> > graph[V];
    addEdge(graph, 0, 1, 1);
    addEdge(graph, 0, 4, 5);
    addEdge(graph, 1, 1, 2);
    addEdge(graph, 1, 2, 3);
    addEdge(graph, 1, 3, 4);
    addEdge(graph, 1, 4, 2);
    addEdge(graph, 3, 2, 2);
    addEdge(graph, 3, 4, 3);
    for (int i = 0; i < V; i++)
    {
        cout << i;
        for (int j = 0; j < graph[i].size(); j++)
        {
            cout << "->" << graph[i][j].first << "," << graph[i][j].second;
        }
        cout << endl;
    }
    return 0;
}
```

Incidence matrix: Matrix where each column represents an edge connected to two vertices.
Matrix of V*E. Possible values for undirected graph:0,1. Possible values for directed graph:-1,0,1

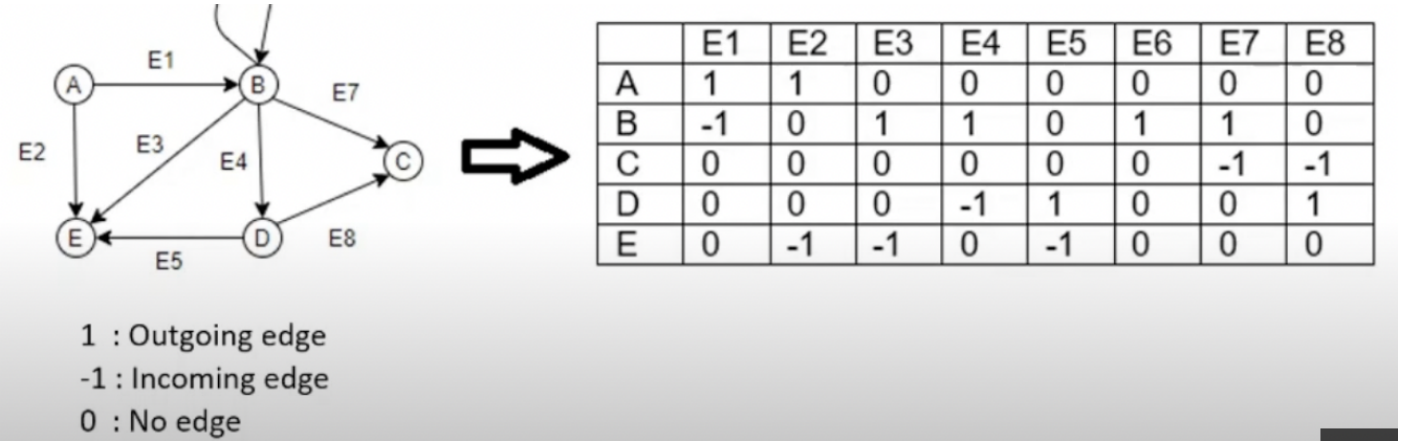
Undirected graph:



	E1	E2	E3	E4	E5	E6	E7	E8
A	1	1	0	0	0	0	0	0
B	1	0	1	1	0	1	1	0
C	0	0	0	0	0	0	1	1
D	0	0	0	1	1	0	0	1
E	0	1	1	0	1	0	0	0

For weighted just instead of 1, write weight.

Directed graph:



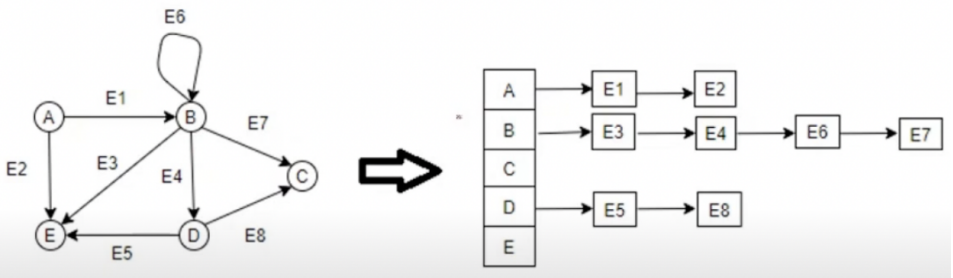
	E1	E2	E3	E4	E5	E6	E7	E8
A	1	1	0	0	0	0	0	0
B	-1	0	1	1	0	1	1	0
C	0	0	0	0	0	0	-1	-1
D	0	0	0	-1	1	0	0	1
E	0	-1	-1	0	-1	0	0	0

Disadvantages:

1. It uses O(VE) space as opposed to O(V^2) in adjacency matrix
2. Checking if a node is related to some other node is O(E). (Say we need to find edge connecting B and C, so we will traverse the row of B and C and where we find common that will be the edge)
3. Traversing a node's adjacencies is O(E)

```
#include <bits/stdc++.h>
using namespace std;
#define V 5
#define E 8

void addEdge(int mat[V][E], int u, int v, int e)
{
    mat[u][e]=1;
    if(u!=v)
        mat[v][e]=-1;
}

int main()
{
    int mat[V][E] = {0};
    addEdge(mat, 0, 1, 0);
    addEdge(mat, 0, 4, 1);
    addEdge(mat, 1, 4, 2);
    addEdge(mat, 1, 3, 3);
    addEdge(mat, 1, 1, 5);
    addEdge(mat, 1, 2, 6);
    addEdge(mat, 3, 4, 4);
    addEdge(mat, 3, 2, 7);
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < E; j++)
        {
            cout << mat[i][j]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

Incidence List for Directed graph:



Through this you won't be able to make the graph because like you know A is connected with E1 but you don't know to who. So you need to store it like E1,B. So extra space needed.

```
#include <bits/stdc++.h>
using namespace std;
#define V 5

void addEdge(vector<pair<int, int> > graph[V], int v, int end, int edge)
{
    graph[v].push_back(make_pair(edge, end));
}

int main()
{
    vector<pair<int, int> > graph[V];
    addEdge(graph, 0, 1, 0);
    addEdge(graph, 0, 4, 1);
    addEdge(graph, 1, 4, 2);
    addEdge(graph, 1, 3, 3);
    addEdge(graph, 1, 1, 5);
    addEdge(graph, 1, 2, 6);
    addEdge(graph, 3, 4, 4);
    addEdge(graph, 3, 2, 7);
    for (int i = 0; i < V; i++)
    {
        cout << i;
        for (int j = 0; j < graph[i].size(); j++)
        {
            cout << " -> " << graph[i][j].first << "," << graph[i][j].second;
        }
        cout << endl;
    }
    return 0;
}
```

Graph Traversals:

1. BFS: Breadth First Search
2. DFS: Depth First Search

Breadth First Search: Queue

Always optimal solution

Time complexity: O(V+E)

Time complexity: O(V+E)

Space complexity: O(V), O(b^d)

b=branching factor, avg. outdegree

d=distance from start node

```
#include<bits/stdc++.h>
using namespace std;
#define V 7

void addEdge(vector<int> graph[V],int u, int v){
    graph[u].push_back(v);
    graph[v].push_back(u);
}

void bfs(vector<int> graph[V], int start){
    vector<bool> visited(V,false);
    queue<int> q;
    q.push(start);
    visited[start]=true;
    while(!q.empty()){
        int v=q.front();
        cout<<v<<" ";
        q.pop();
        for(auto i=graph[v].begin();i!=graph[v].end();i++){
            if(!visited[*i]){
                q.push(*i);
                visited[*i]=true;
            }
        }
    }
}

int main(){
    vector<int> graph[V];
    addEdge(graph,0,1);
    addEdge(graph,0,3);
    addEdge(graph,1,2);
    addEdge(graph,2,3);
    addEdge(graph,2,6);
    addEdge(graph,3,4);
    addEdge(graph,4,5);
    addEdge(graph,5,6);
    bfs(graph,0);
    return 0;
}
```

Applications:

1. Peer to Peer Network (Torrent wagers)
2. Social Networking Websites (Linkedin mein 1st, 2nd, 3rd jo aata hai)
3. GPS Navigation systems (like find nearby things)
4. Path finding (Finding shortest path)
5. Broadcasting in Network
6. Garbage collection

Depth First Search: Stack

Not optimal solution always

Time complexity: O(V+E)

Space complexity: O(V), O(bd)

b=branching factor, avg. outdegree

d=distance from start node

```
#include<bits/stdc++.h>
using namespace std;
#define V 7

void addEdge(vector<int> graph[V], int u, int v){
    graph[u].push_back(v);
    graph[v].push_back(u);
}

void dfs_i(vector<int> graph[V], int start){
    vector<bool> visited(V,false);
    stack<int> s;
    s.push(start);
    visited[start]=true;
    while(!s.empty()){
        int v=s.top();
        cout<<v<<" ";
        s.pop();
        for(auto i=graph[v].begin();i!=graph[v].end();i++){
            if(!visited[*i]){
                s.push(*i);
                visited[*i]=true;
            }
        }
    }
}

void dfs_r(vector<int> graph[V], vector<bool>& visited, int start){
    cout<<start<<" ";
    visited[start]=true;
    int v=start;
    for(auto i=graph[v].begin();i!=graph[v].end();i++){
        if(!visited[*i])
            dfs_r(graph,visited,*i);
    }
}

int main(){
    vector<int> graph[V];
    addEdge(graph,0,1);
    addEdge(graph,0,3);
    addEdge(graph,1,2);
    addEdge(graph,2,3);
    addEdge(graph,2,6);
    addEdge(graph,3,4);
    addEdge(graph,4,5);
    addEdge(graph,5,6);
    dfs_i(graph,0);
    cout<<endl;
    vector<bool> visited(V,false);
```

```
    dfs_r(graph,visited,0);
    return 0;
}
```

Applications:

1. Topological sorting
2. Scheduling problems
3. Cycle detection in graph
4. Solving puzzles with only one solution like maze

Dijkstra Algorithm: Algorithm for finding shortest path between nodes in a graph
Used in cellular networks, GPS navigation systems, routing protocols etc.

Principle: Shortest possible path from the source has to come from one of the shortest paths already discovered.

Advantages:

1. Fast and simple algorithm

Disadvantages:

1. Need to store large parts of the graph in memory
2. Fails for negative edge weights

```
#include <bits/stdc++.h>
using namespace std;

vector<int> dijkstra(vector<vector<pair<int, int> > > graph, int start)
{
    vector<int> dist(graph.size(), INT_MAX);
    priority_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int, int> > > pq;
    pq.push(make_pair(start, 0));
    dist[start] = 0;
    while (!pq.empty())
    {
        int u = pq.top().first;
        pq.pop();
        for (int i = 0; i < graph[u].size(); i++)
        {
            int v = graph[u][i].first;
            int weight = graph[u][i].second;
            if (dist[v] > dist[u] + weight)
            {
                dist[v] = dist[u] + weight;
                pq.push(make_pair(v, weight));
            }
        }
    }
    return dist;
}

void addEdge(vector<vector<pair<int, int> > > &graph, int u, int v, int w)
{
```

```
{
    graph[u].push_back(make_pair(v, w));
    graph[v].push_back(make_pair(u, w));
}

int main()
{
    vector<vector<pair<int, int> > > graph(9, vector<pair<int, int> >(9));
    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 7, 8);
    addEdge(graph, 1, 2, 8);
    addEdge(graph, 1, 7, 11);
    addEdge(graph, 2, 3, 7);
    addEdge(graph, 2, 8, 2);
    addEdge(graph, 2, 5, 4);
    addEdge(graph, 3, 4, 9);
    addEdge(graph, 3, 5, 14);
    addEdge(graph, 4, 5, 10);
    addEdge(graph, 5, 6, 2);
    addEdge(graph, 6, 7, 1);
    addEdge(graph, 6, 8, 6);
    addEdge(graph, 7, 8, 7);
    vector<int> dist = dijkstra(graph, 0);
    for (int i = 0; i < dist.size(); i++)
        cout << i << "   " << dist[i]<<endl;
    return 0;
}
```

Topological Sort: Linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Only possible for DAG (Directed Acyclic Graph)

Applications:

1. Job Scheduling by operating systems
2. Maven Dependency Resolution
3. Order of load tables with foreign keys in database

Kahn's Algorithm:

```
#include<bits/stdc++.h>
using namespace std;
#define V 5

void addEdge(vector<int> graph[V],int u, int v){
    graph[u].push_back(v);
}

vector<int> kahn(vector<int> graph[V]){
    vector<int> result;
    queue<int> q;
    vector<int> indegree(V,0);
    for(int i=0;i<V;i++){
        for(int j=0;j<graph[i].size();j++){
            indegree[graph[i][j]]++;
        }
```

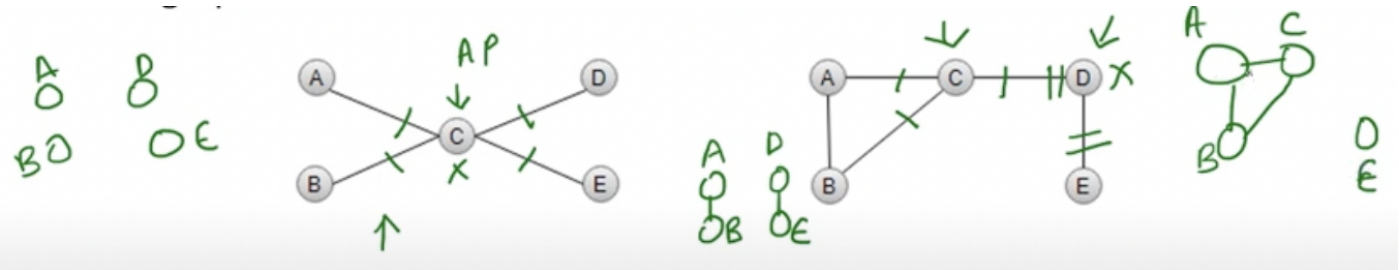
```
        }
    }
    for(int i=0;i<indegree.size();i++){
        if(indegree[i]==0)
            q.push(indegree[i]);
    }
    while(!q.empty()){
        int v=q.front();
        q.pop();
        result.push_back(v);
        for(auto i=graph[v].begin();i!=graph[v].end();i++){
            indegree[*i]--;
            if(indegree[*i]==0)
                q.push(*i);
        }
    }
    return result;
}

int main(){
    vector<int> graph[V];
    addEdge(graph,0,1);
    addEdge(graph,0,2);
    addEdge(graph,1,2);
    addEdge(graph,1,3);
    addEdge(graph,2,3);
    addEdge(graph,2,4);
    vector<int> result=kahn(graph);
    if(result.size()==V){
        for(int i=0;i<V;i++){
            cout<<result[i]<<" ";
        }
    }
    else{
        cout<<"Topological sort not possible!";
    }
    return 0;
}
```

Modified DFS:

```
#include<bits/stdc++.h>
using namespace std;
#define V 5

void addEdge(vector<int> graph[V],int u, int v){
    graph[u].push_back(v);
}

void dfs(vector<int> graph[V], vector<bool>& visited, vector<int>& result, int node){
    visited[node]=true;
    int v=node;
    for(auto i=graph[v].begin();i!=graph[v].end();i++){
        if(!visited[*i]){
            dfs(graph,visited,result,*i);
        }
    }
    result.push_back(node);
```

```
}

int main(){
    vector<int> graph[V];
    vector<bool> visited(V,false);
    addEdge(graph,0,1);
    addEdge(graph,0,2);
    addEdge(graph,1,2);
    addEdge(graph,1,3);
    addEdge(graph,2,3);
    addEdge(graph,2,4);
    vector<int> result;
    dfs(graph,visited,result,0);
    if(result.size()==V){
        for(int i=V-1;i>=0;i--){
            cout<<result[i]<<" ";
        }
    }
    else{
        cout<<"Topological sort not possible!";
    }
    return 0;
}
```

Articulation Points: A vertex is an articulation point if removing it and all the edges associated with it, increases the number of connected components in the graph.



Biconnectivity: A connected undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph. Basically no articulation points.

Brute Force Algorithm: Remove all vertices one by one and see if removal of a vertex causes disconnected graph.
Algo:
result=[]
for each v in V
    remove v from graph
    if checkConnected()!=true
        append v to the result

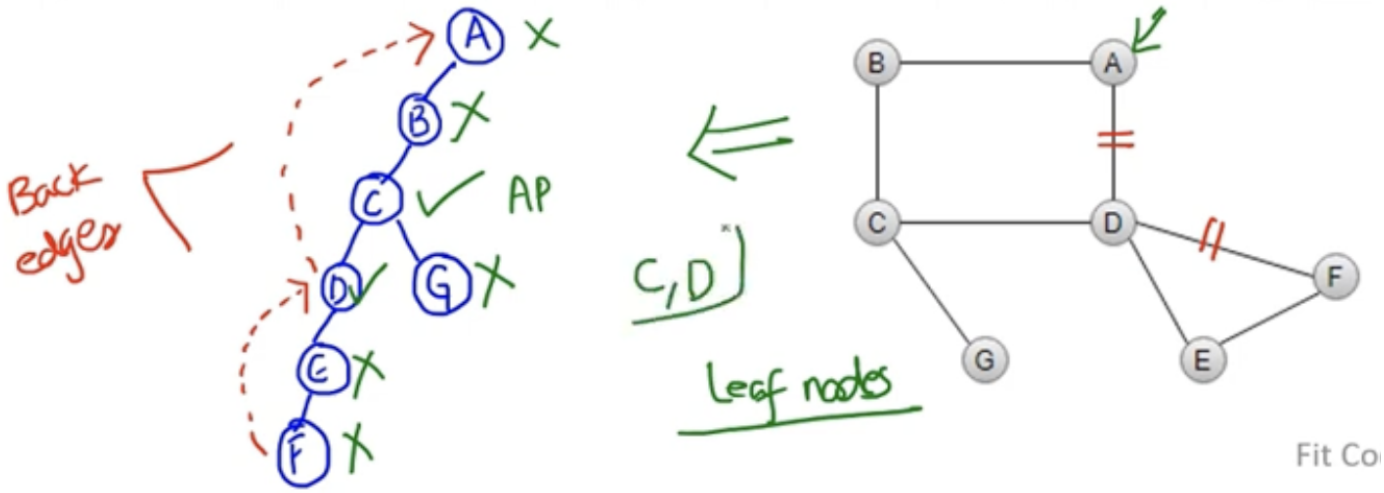
checkConnected()
    use BFS or DFS traversal

Tarjan's Algorithm:

In DFS tree, a vertex u is articulation point if one of the conditions is true:

1. u is the root of DFS tree and it has at least 2 children
2. u is not root of DFS tree and it has a child v such that no vertex in subtree of v is connected to ancestor of v using a backedge

Backedge are edges which are missing in DFS tree but present in the graph



For each node compute 2 things:

1. discovery time
2. low time: time of the earliest node that can be reached from subtree(u) i.e. low[u]=min(disc[u],disc[v],low[v])