Todays Content

- → Graph Intro
- → Classification
- → Input
- → BFS

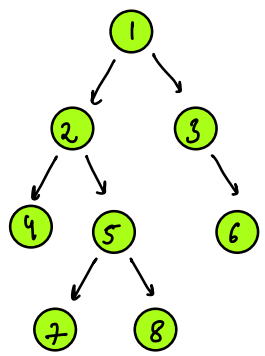
Introduction to Graphs

Graph: Connection of nodes & edges.

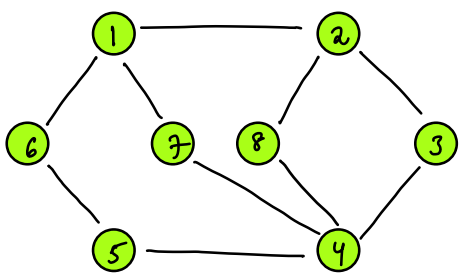
Ex1: Tree

N = 8

E = 7



Graph : N = 8 edges = 10



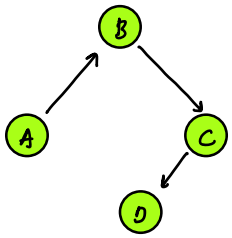
Main diff between Trees & graphs

1. Tree is a hirarchal data structure
2. In Tree: Edges = Nodes - 1

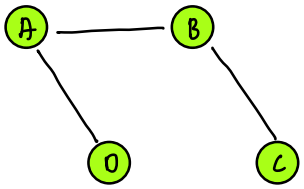
   In Graph: above condition may or may not satisfy.

Classification of Graphs

directed graph



Undirected



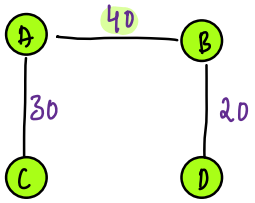
facebook:

Instagram:

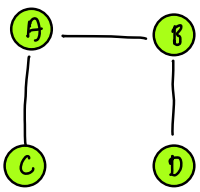
Twitter / LinkedIn

weighted ——:

- distance
- time
- mutual friends
- Cost
- profit/loss

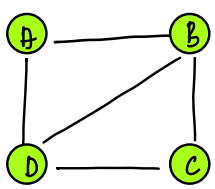


unweighted

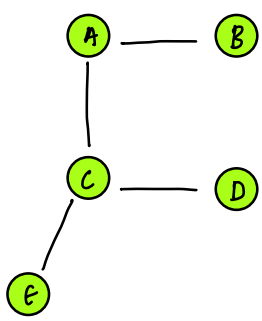


Note: Weight is assigned to an edge.

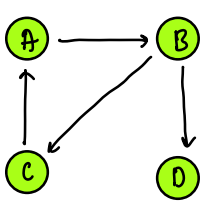
Undirected Cyclic



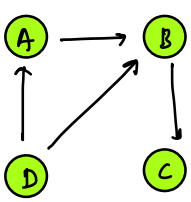
Undirected Acyclic



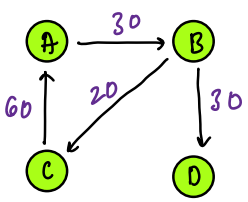
directed cyclic



Directed Acyclic graph: DAG



Note: Graph can be combination of classification



- directed ✓
- weighted ✓
- Cyclic ✓

How is graph given as input ?

: Graph is Collection of Nodes & Edges

Eg: Nodes & Edges of directed Graph.

Input format:

1st line: N E

N: no: of Nodes

E: no of Edges

followed by E line

: U & V

: Indices Edge Between node U & V

If weighted Edge

: U V W

Edge between U & V with Weight W

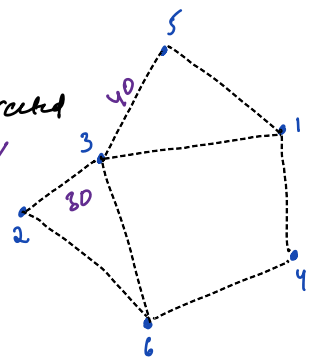
N E

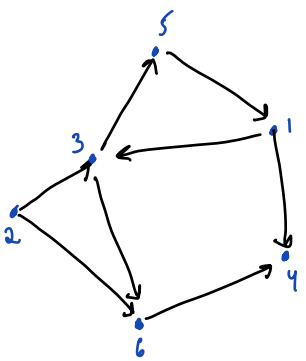
6 : nodes 8 : Edges

Edges: U V

	u[8]	v[8]	w[8]
u[0]..v[0]	2	3	30
u[1]..v[1]	3	5	40
u[2]..v[2]	1	4	20
u[3]..v[3]	6	4	10
u[4]..v[4]	2	6	6
u[5]..v[5]	5	1	80
u[6]..v[6]	1	3	20
u[7]..v[7]	3	6	50

If directed u → v



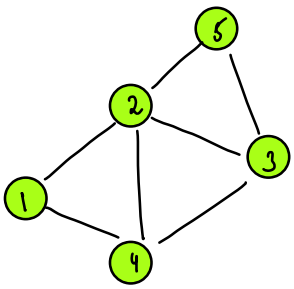


Question: Following inf is given in Question

a) Directed / Undirected

b) Weighted / Unweighted

Storing Undirected Graph:



Input:

N	E
5	7

u	v
1	4 ✓
2	5 ✓
3	2 ✓
4	3 ✓
2	4 ✓
3	5 ✓
1	2 ✓

Way1: Adj mat:

int mat[6][6] = 0

	0	1	2	3	4	5
0						
1		0	1	0	1	0
2		1	0	1	1	1
3		0	1	0	1	1
4		1	1	1	0	0
5		0	1	1	0	0

// Say N Nodes & E Edges : mat[N+1][N+1]

	unweighted	weighted u v w :
undirected	u ___ v mat[u][v] = 1 mat[v][u] = 1	u ___w___ v mat[u][v] = w mat[v][u] = w
directed	u ⟶ v mat[u][v] = 1	u ⟶w v mat[u][v] = w

TC for adj matrix:

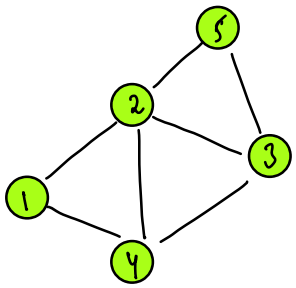
TC: O(E)

SC: (N+1)(N+1) ≈ O(N²)

Disadvantages:

: To much Space Wastage.

: To avoid Space Wastage we go to adj list



`list<int> l[3];` array of lists of 10

3 5 : 2 =

6 7 8 : 3

10 2 9 2 5 : 5

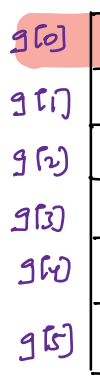
Note: array of list : TODO
Check declaration in your language

Input:

N E
5 6
u : v
1 - 4 ✓
2 - 5 ✓
3 - 2 ✓
4 - 3 ✓
2 - 4 ✓
3 - 5

Way2: Adj list: →

`list<int> g[6]`



g[0] Empty
g[1] 4 → : 1
g[2] 5 3 4 : 3
g[3] 2 4 5 : 3
g[4] 1 3 2 : 3
g[5] 2 3 : 2

Sum of sizes of = 12
= 6 × 2

Input: Directed Weighted Graph

`list<pair<int, int>> g[4]`

N E
3 5
u v w
1 - 2 : 5
1 - 3 : 2
2 - 3 : 4
2 1 6
3 1 7

v w v w
<2 5> <3, 2>
<3, 4> <1, 6>
<1, 7>

// Given N & E Edges

a) Unweight graph: `list<int> g[N+1]`

b) Weight graph: `list<pair<int, int>> g[N+1]`

	unweighted	weighted
undirected	u ——— v g[u].add(v) g[v].add(u)	u —w— v g[u].add(pair<v, w>) g[v].add (pair<u, w>)
directed	g[u].add(v)	u —w→ v g[u].add (pair<v, w>)

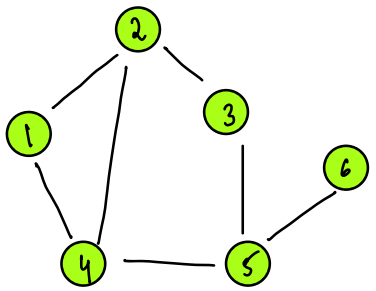
TC: O(E) : To add 1 element in list : O(1), E edges : O(E)

Sc: In Undirected : For every edge : 2 value = 2E + N = O(N+E)

In Directed : For every edge : 1 value = E + N = O(N+E)

IQ: Given a undirected graph & Source Node & Dest Node
Check if node can be visited from Source Node?

Graph: S/1 D/6 :



Input:

N	E
6	7
u[7]	v[7]
1	2
1	4
2	4
2	3
3	5
5	6
4	5

Construct Adj list S=1 D=6 Check path:?

list<int> g[7]

g[0]
g[1] 2 4
g[2] 1 4 3
g[3] 2 5
g[4] 1 2 5
g[5] 3 6 4
g[6] 5

Note: Each node should be added only once
↓
Check it using bool ch[ ]

Idea: bool ch[7] =

0	1	2	3	4	5	6
F	F	F	F	F	F	F
	T	T	T	T	T	T

ch[6] == True: visited.

Datastruct: Queue

g[4][0]: 1
g[4][1]: 2
g[4][2]: 5

Idea: BFS: Breath First Search?

1. Delete front ele from Queue: u
2. Iterate on nodes connected to u?
   1. Iterate on g[u] to get nodes connected to u
   2. Add a particular node in Queue if it's not visited, mark as visited.
3. Repeat till Queue is Empty
4. Check visited of destination is True/False

Param: Nodes N, Edges E, Edges: u[E] ↔ v[E] S D

```
bool bfs ( int N, int E, int u[E], int v[E], int S, int D){

  Step1: Created adj list  TC: O(E)  SC: O(N+E)

      list<int> g[N+1]
      i=0; i<E; i++){
        //ith edge: u[i] ——— v[i]
        int a = u[i]  int b = v[i]
        g[a].add(b)
        g[b].add(a) // If directed remove
      }                                          TC: O(N+E)
                                                 SC: O(N+E)

  Step2: Apply BFS  TC: O(N+E)  O(N+N)

      bool vis[n+1] = False;  int l[n+1] = -1;  int p[n+1] = -1
      Queue<int> q
      q.insert(s)  vis[s] = True  l[s] = 0  p[s] = -1
      while( q.size()>0){
          int u = q.front();
          q.delete() // delete front ele que
          // Get nodes connected to u ?
          // Iterate on list g[u]
            i=0; i< g[u].size(); i++){
              int v = g[u][i]      //u ——— v
              if( vis[v] == false){
                  q.insert(v)
                  vis[v] = True  l[v] = l[u]+1  p[v] = u
              }
          }
      return vis[D] // l[D] // using p[] get path S→D //
}
```

Worst Case: We delete every node.

u	g[u]	Total Iter
1	delete(1) : 1 + iterate on g[1] :	1 + g[1].size()
2	delete(2) : 1 + iterate on g[2] :	1 + g[2].size()
3	delete(3) : 1 + iterate on g[3] :	1 + g[3].size()
⋮	⋮	
n	delete(n) : 1 + iterate on g[n] :	1 + g[n].size()

(Total Adj list Element)

undirected : 2E

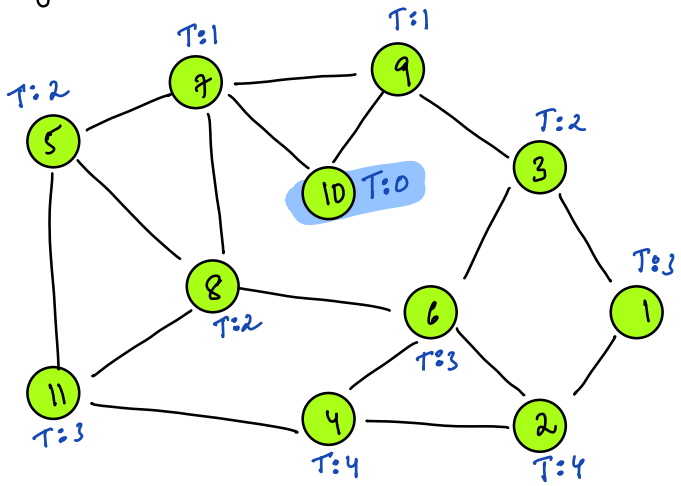
directed : E

: O(E)

n + O(E) = O(N + E)

len of Shortest Path:

Tracing S = 10 D = 2 :

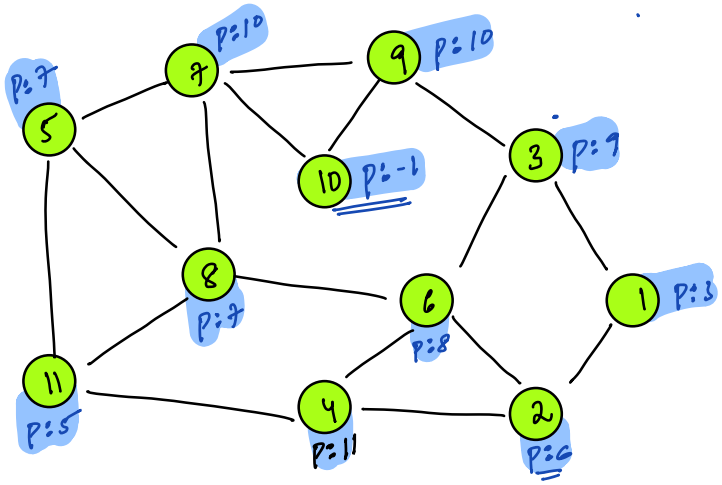


level: 0 1 2 3 4

10 | 7 9 | 5 3 8 | 6 1 | 11 4 2

Note: Using BFS we can get length of shortest path from S → D

Tracing S=10 D=2:



~~10~~	~~7~~	~~9~~	~~5~~	~~8~~	~~3~~	~~11~~	~~6~~	~~1~~	~~4~~	~~2~~

Note: For every node, store it's parent node.

par[12] =

0	1	2	3	4	5	6	7	8	9	10	11
-1	3	6	9	11	7	8	10	7	10	-1	5

S:10 D:2 par[2] par[6] par[8] par[7] par[10]

D:2 → 6 → 8 → 7 → 10 → -1 Stop

S:10 D:11 par[11] par[5] par[7] par[10]

D:11 → 5 → 7 → 10 → -1 Stop.

11(a) Given S & D // get shortest path from S → D

1) Fill par[] —

2) listofints path

```
n = D
while(n != -1){
    path.insert(n)
    n = par[n]
}
return path;
```