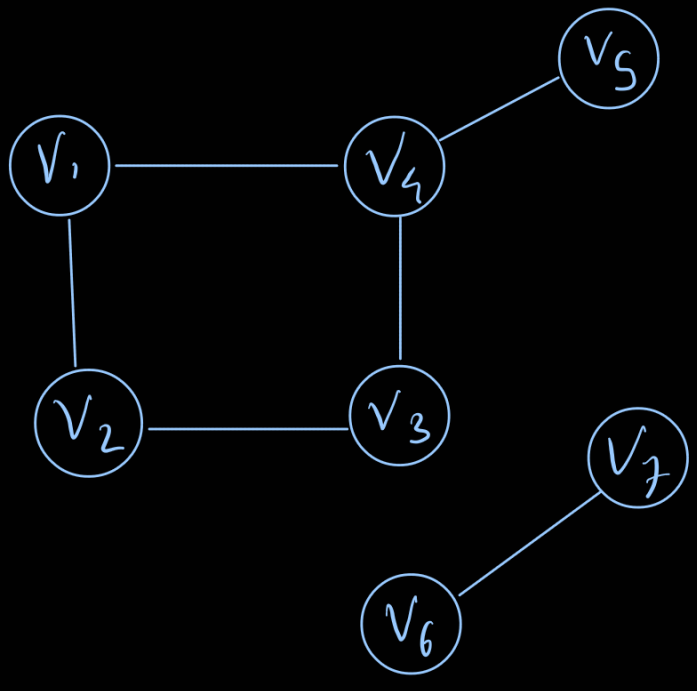

DSA
GRAPHS
SHIVANSH SANTOKI

# What is a Graph?

## Formal:

→ A graph $G = (V, E)$ consists of a vector set $V$ and edge set $E$, such that $E \subseteq V \times V$

Eg:




$|V| = 7$

$|E| = 6$

$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E = \{(v_1, v_2), (v_1, v_4), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_6, v_7)\}$

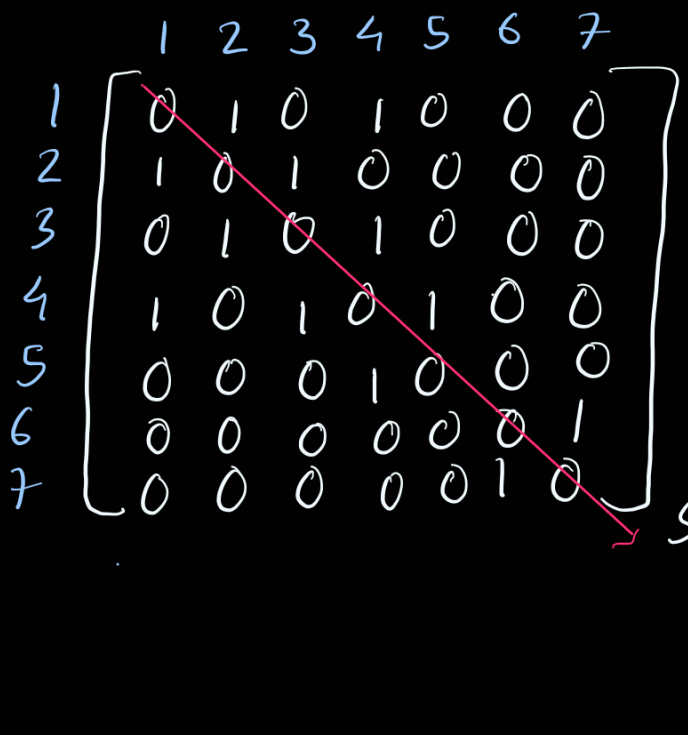
## Adjacency Matrices:

Denotes if ∃ a edge b/w two vertices or not.

If edge b/w $i$ and $j$ nodes → $a_{ij} = 1$

else → $a_{ij} = 0$

∴ In this case, the matrix will be 7×7.




→ What is the maximum number of edges that a graph $G_1 = (V, E)$ on n vertices $(|V| = n)$ can have?

$$|E| = {}^nC_2 = \boxed{\frac{n(n-1)}{2}}$$

(Edge btw every two nodes)

→ How many different graphs are there on the vertex set $V = \{1, 2, \ldots, n\}$

→ (${}^nC_2$ ways of this)

Pick any two vertices from the set, there are 2 possible cases, either 1 or 0.

$$\therefore (2)^{{}^nC_2} = \boxed{(2)^{\frac{n(n-1)}{2}}}$$

Proof: ${}^2C_0 + {}^2C_1 + {}^2C_2 + \cdots + {}^2C_2 = (2)^2$

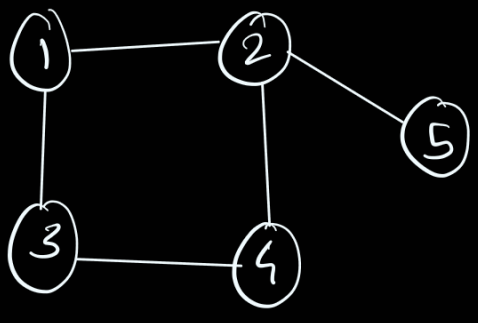
$= (2)^{\frac{n(n-1)}{2}}$

↑ zero edges ↑ one edge

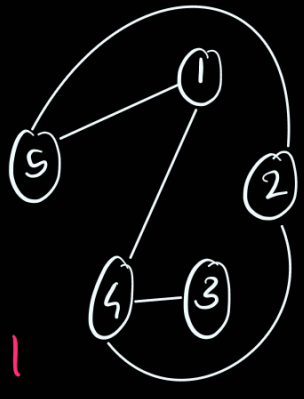
Matrix

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 1 | 0 | 0 | 1 | 1 |
| 3 | 1 | 0 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 0 |
| 5 | 0 | 1 | 0 | 0 | 0 |

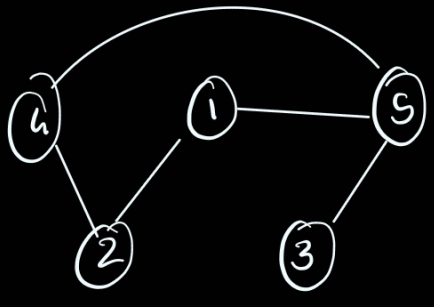
Graph




Eg:

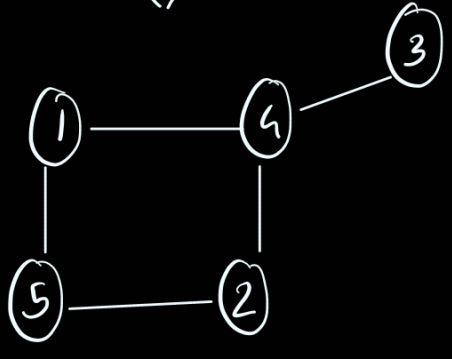


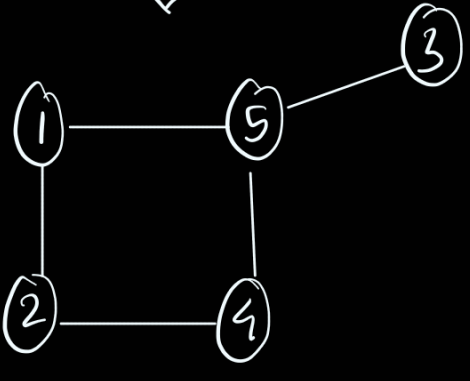

G1




G2

G1 and G2 are isomorphic




Two graphs are isomorphic to each other if the rows & colums of the adjacency matrix of one of them can be exchanged so that these are same as the adjacency matrix of other.

# NOTE:

Given two graphs $G_1$ and $G_2$, both with $n$ vertices, check if they are isomorphic or not.

$TC(\text{Best Algo}) = \boxed{2^{O(\log n)^3}}$ (Babai, 2016)

---

## ✗ 3 Ways of Representing Graph:

(A) Adjacency Matrix [nxn matrix, 2D array]

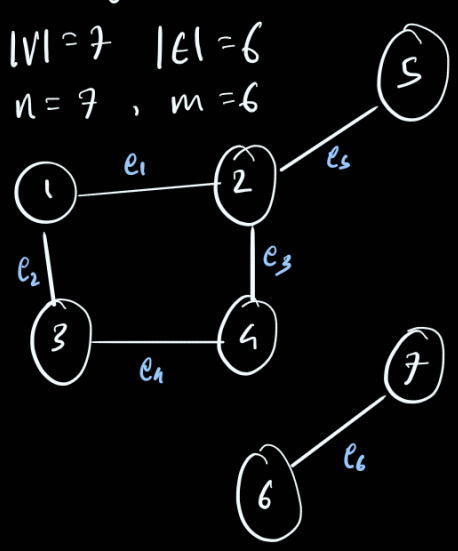
(B) Incidence Matrix [nxn matrix, 2D array]

(C) Adjacency List [n-linked lists, Space: 2n]

let the graph be:

$|V| = 7 \quad |E| = 6$

$n = 7, \quad m = 6$




### (A) Adjacency Matrix:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 3 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

If $(v,u) \in E$

then,

$Adj[v][u] = 1$

Space = $n \times n = 7 \times 7$

## (B) Incidence matrix:

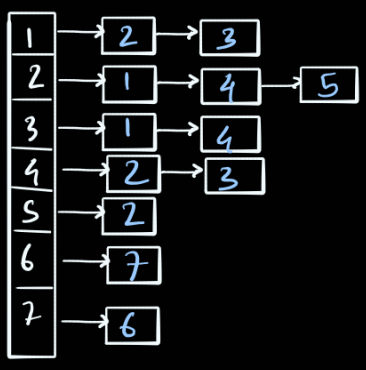
| | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ | $e_6$ |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 1 | 1 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 |

Space:
$n \times m$
$= 7 \times 6$

Disadvantage:
If $n = 20,000$, then 19,998 elements in each column will be zero.

$$m \leq \frac{n(n-1)}{2}$$

## (C) Adjacency List:




Space: $2E = 2 \times 6 = 12$

| | Adjacency List | Adjacency Matrix |
|---|---|---|
| Space Complexity | $O(2m)$ | $O(n^2)$ |
| $(v,u) \in E$? | $O(n)$ | $O(1)$ |
| All neighbours of a vertex | $O(n)$ | $O(n)$ |
| All edges | $O(n+m)$ | $O(n^2)$ |
| Insert edge | $O(1)$ | $O(1)$ |
| Delete edge | $O(n)$ | $O(1)$ |

**Neighbour:**
A vertex $v$ is said to be a neighbour of $u$, if $(v,u) \in E$

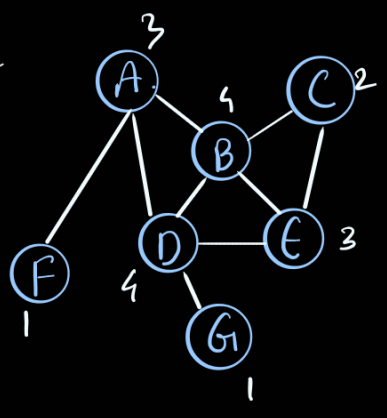
**Degree:**
The degree of a vertex is the number of neighbours it has.

$$0 \leq deg(v) \leq n-1$$

# ※ Handshaking Lemma :

If each person at a party shakes hands with those that the person is friends with, then the total number of handshakes is equal to half the [(cummulative total of) the number of handshakes each person made].

Eg:




$deg(A) + deg(B) + \cdots + deg(G)$

$= 3 + 4 + 2 + 4 + 3 + 1 + 1$

$= \boxed{18}$

$n(Edges) = \boxed{9}$

$\therefore \boxed{n(E) = \frac{\Sigma deg(\forall V)}{2}}$

Fact: No. of people, who shakes odd number of hands is even.

M-1

Proof: let $n = |V|$ and, $m = |E|$

As each edge contribute to the degrees of two vertices,

$\Sigma deg(V_i) = 2m$

let $x$ = no. of vertices with odd degree.

$\therefore \Sigma deg = (n-x)(even) + (x)(odd)$

$\therefore 2m = even + x \times odd$

$\therefore even = even + x \times odd$

$\therefore even = x \times odd$

$\therefore \boxed{x = even}$

M-2 Empty Graph: 0 is even (Base Case)

There are 3 cases:

Case (1): A (odd), B (odd) ⇒ A — B (even, even) ※odd -= 2

Case (2): A (even), B (even) ⇒ A — B (odd, odd) ※odd += 2

Case (3): A (odd), B (even) ⇒ A — B (even) ※odd += 0

} Inductive Step

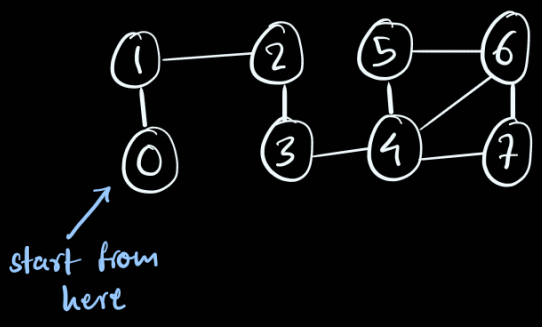
## ※ Breadth - First - Search (BFS):

- Start with a vertex A.
- Then explore the neighbours of A.
- Repeat this for each of the neighbours of A.
  And so on...
- Keep exploring until there is nothing left to explore.

## Pseudo Code:

```
BFS(A)
{
    Push (null, A) to queue Q.
    while(Q is nonempty)
    {
        pop(p, v) from Q
        if(visited(v) == FALSE)
        {
            visited(v) ← TRUE
            parent(v) ← p
            for each edge (v, w)
                push (v, w) to Q
        }
    }
}
```

**Eg:** Let |V| = 8, |E| = 9




visited:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

BFS Traversal : ans

1) queue : (null, 0) pop

ans : 0

visited :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | | |

2) queue : (0, 1) pop

ans : 0 1

visited :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

3) queue : (1, 2) pop

ans : 0 1 2

visited :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

4) queue : (2, 3) pop

ans : 0 1 2 3

visited :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

5) queue : (3, 4) pop

ans : 0 1 2 3 4

visited :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

6) queue : (4, 5), (4, 6), (4, 7) pop

ans : 0 1 2 3 4 5

visited :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

7) queue : (4, 6), (4, 7) pop

ans : 0 1 2 3 4 5 6

visited :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

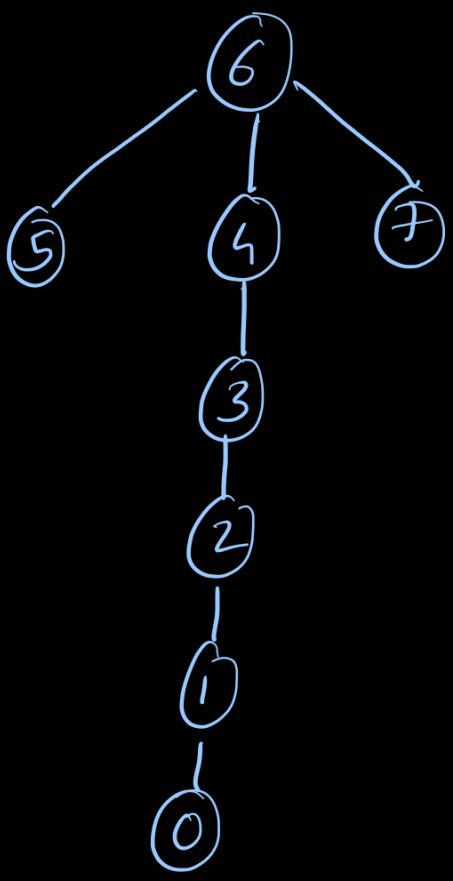
8) queue : (4, 7) pop

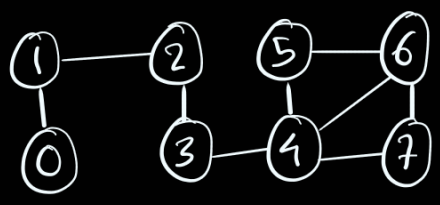
ans : 0 1 2 3 4 5 6 7

Q empty → loop terminates.

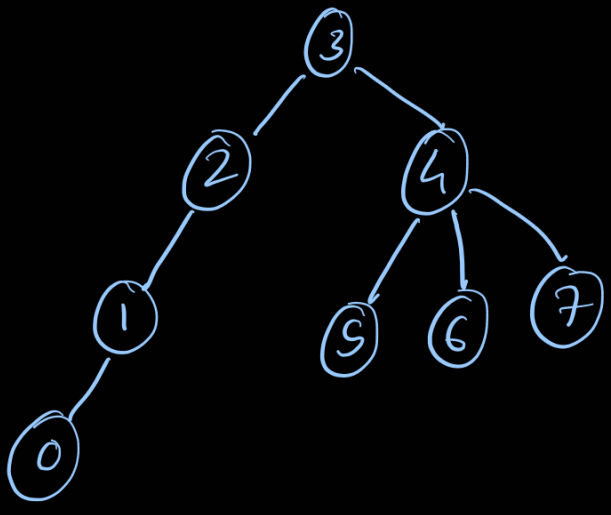
Q: What will be the BFS tree for

(i) BreadthFirstSearch(6):

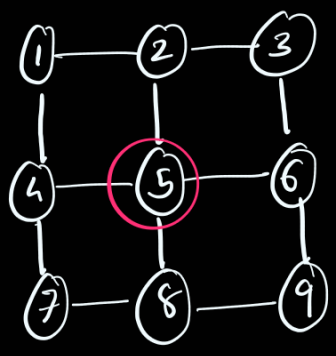



(ii) BreadthFirstSearch(3):



**Q:** Given graph:




$|V| = 9$

$|E| = 12$

**What is a forward edge:**

An edge from vertex to it's parents

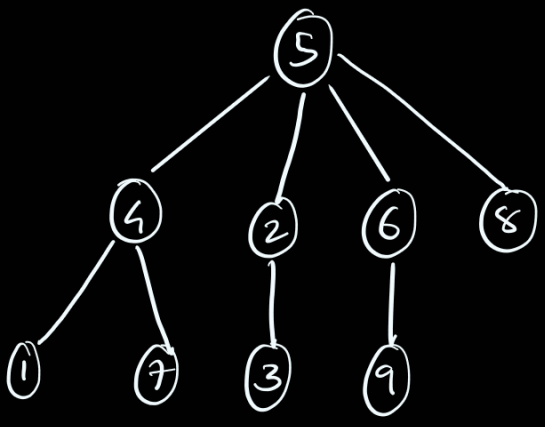
**Cross edge:**

Edge b/w two vertices at the same level

**Back edge:**

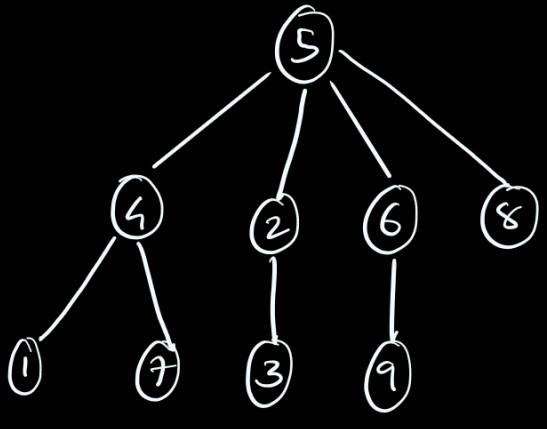
Edge from a vertex at a higher level to a vertex at a lower level.

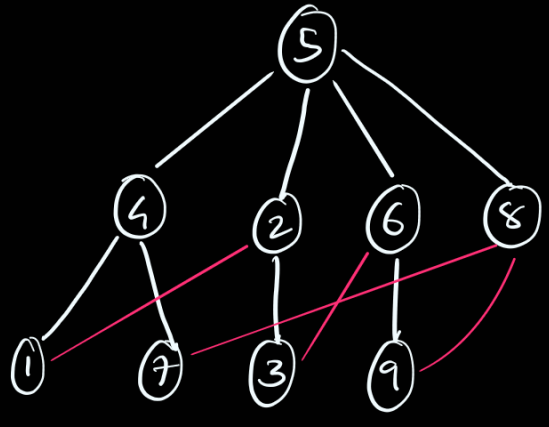
**BFS Tree:**




forward edges = 8

(f)




Cross edges : 0

(c)

back edge : 4

(b)

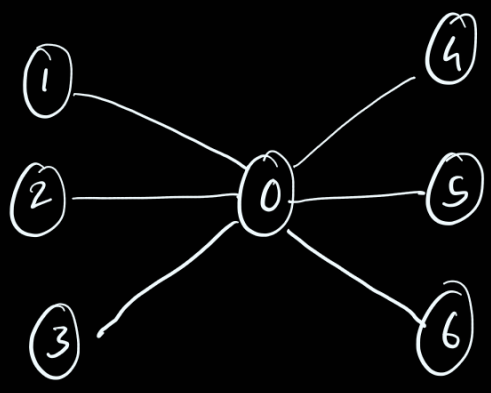
$f + c + b = |E|$ ✓

$8 + 0 + 4 = 12$

→ A graph which does not have any cross edges is a bipartite graph.

→ No odd length cycle.

Eg: Given Tree:

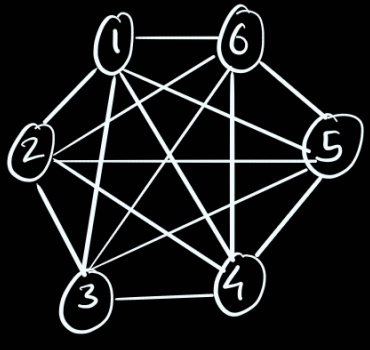



Here, every edge is a forward edge.

A graph which has 0 cross and back edges is a tree. (All edges are forward edges)

$|V| = 7 \quad |E| = 6$

Eg: Complete Graphs:



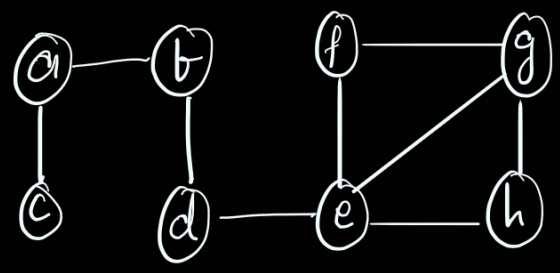
# ※ Depth First Search

Pseudo Code:

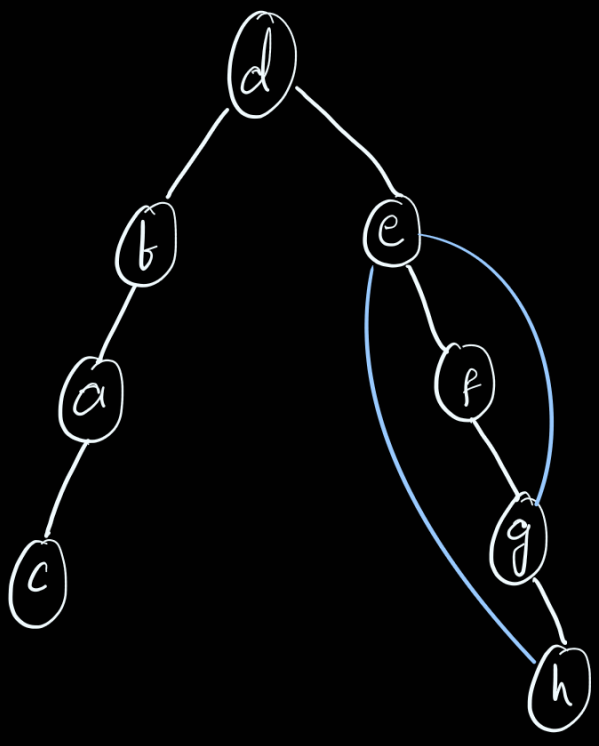
```
DFS(s)
{
    push(null, s) to stack S
    while(S is not empty)
    {
        pop(p, v) from S
        if(visited(v) == FALSE)
        {
            visited(v) ← TRUE
            parent(v) ← p
            for(each (v, w) ∈ E) {
                push(v, w) to S
            }
        }
    }
}
```

→ Look at a neighbour of S.
→ Then, look at a neighbour of neighbour of S.
→ Keep traversing in this way and terminate when everything is traversed.

Eg: Consider the following graph:



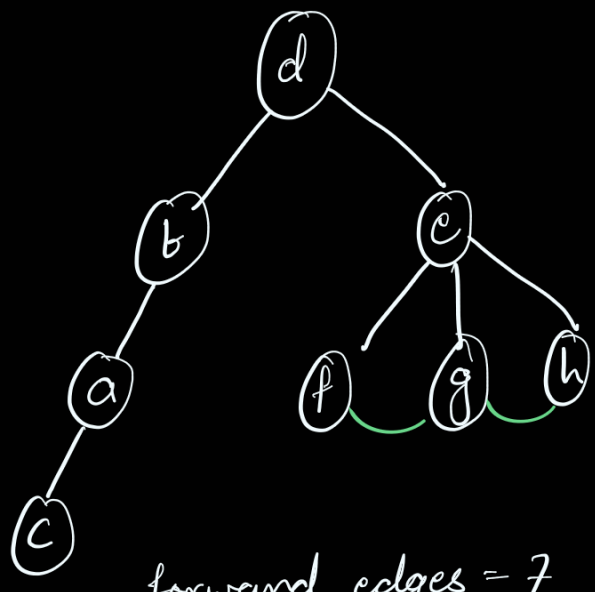

DFS(d)




forward = 7

back edges = 2

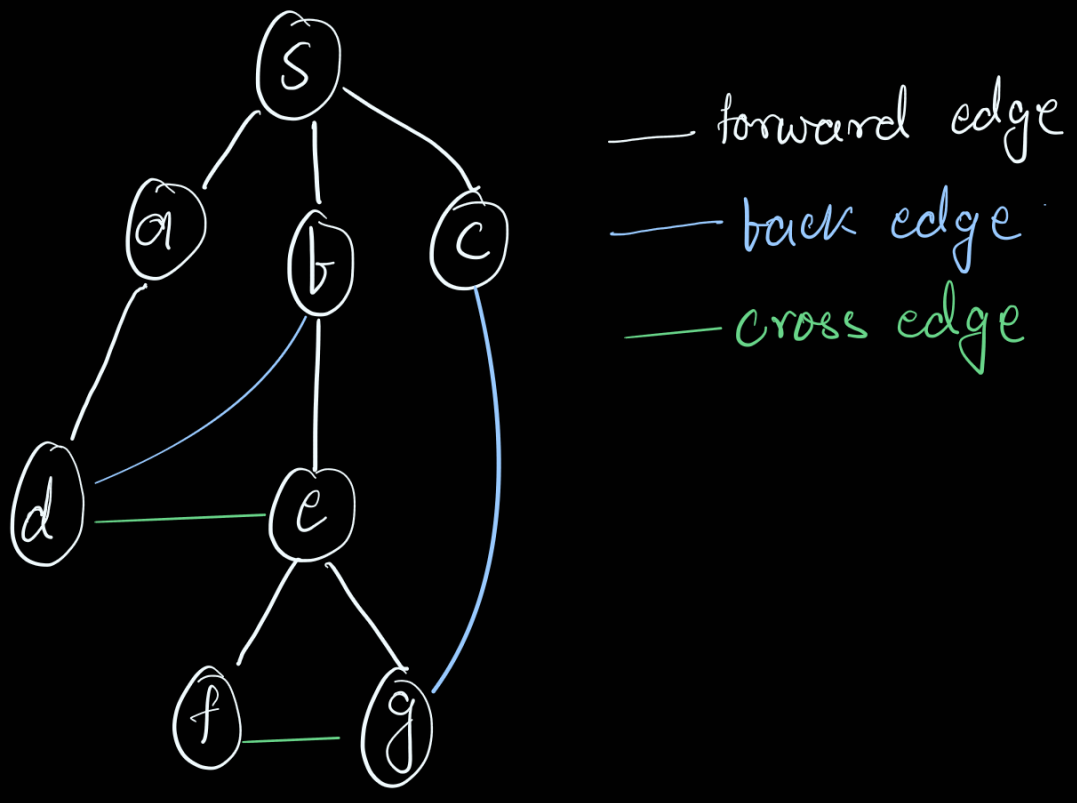
BFS(d)




forward edges = 7

cross edges = 2

Q. Construct the graph for which this is a BSF traversal. You may add more edge if you want.




Facts:

1) Let v be a vertex in level i of a BFS traversal of a graph.
Then, all its neighbours are in one of the level: {i-1, i, i+1} i.e

2) If ∃ an edge b/w vertex at level i and level j, then $|i-j| \leq 1$.

Hence, the given BFS traversal is wrong.

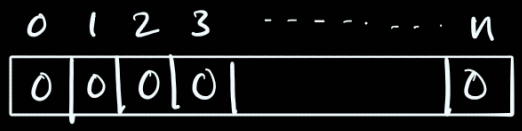
NOTE: The level of a vertex in it's BFS traversal from S is it's distance from s. (only true for BFS).

## Another Fact:

In $O(|V|+|E|)$, it is possible to compute distances of all vertices from s.

**Proof:** Do BFS traversal. Let $|V|=n$, $|E|=m$

let Dist :




Let $s=1$,

then the array "Dist" will store the distance of i from 1.

→ Initially push(1,0)

[vertex, level]

then traverse all of its !visited neighbours:

for each neighbour $(k)$ of (s, level):

```
if !visited(k)
    Dist[k] = level + 1
    push(k, level + 1)
    visited(k) = 1
```

# ※ Types of Graph :

| Type | Definition |
|---|---|
| ① Empty Graph | A graph with no edges |
| ② Complete Graph | A graph in which all pairs of vertices are edges |
| ③ Path | A chain of connected vertices without repetition |
| ④ Cycle | A loop of connected vertices without repetition |
| ⑤ Tree | ∃ exactly one path b/w every pair of vertices. |
| ⑥ Bipartite Graph → | A graph whose vertices can be partitioned into two sets, L & R such that, every edge has one end-point in L and one end-point in R. |
| ⑦ Regular Graph | |
| ⑧ Planar Graph | |

NOTE: While doing BFS traversal on a Bipartite graph, any two vertices at the adjacent level are in different sets.

A graph is a Bipartite graph iff it has no cross edge.

## ※ Bipartite Graph:

→ A graph is bipartite iff its BFS traversal has no cross-edge.

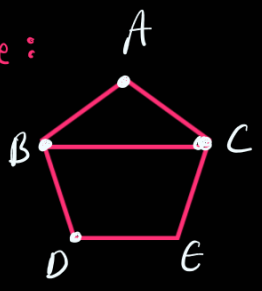
→ A graph is bipartite iff it has no odd-length cycle.

## ※ Cycles in BFS:

→ Odd cycles ⟷ cross edge → Try to prove it

→ Even cycle ⟵ back edge

└→ Converse is not true:
Counterexample:




## ※ Trees

→ No odd cycles
→ No even cycles
→ Only forward edges.
→ Every tree is a bipartite graph.

## ※ Disconnected Graphs:

→ A graph is disconnected iff it contains two vertices u and v such that there is no path between u and v.

→ The empty graph on $n \geq 2$ vertices is disconnected.

## ※ Graph Isomorphism:

→ Two graphs are isomorphic if one of them can be relabeled such that their adjacency matrix are identical.

→ To show that graphs are NOT isomorphic, show one property of one graph that does not exist in the other graph.

# Undirected Graphs

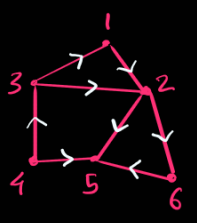
- $G = (V, E)$
- $E \subseteq V \times V$
- $(v, v) \notin E$
- $(u, v) \in E \iff (v, u) \in E$
- $A(i)(j) = A(j)(i)$

  [Adjacency matrix is symmetric]

# Directed Graphs

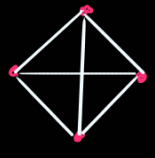
- $G = (V, E)$
- $E \subseteq V \times V$
- $(v, v) \notin E$

Eg:

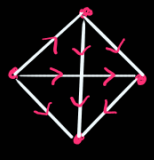



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 1 |
| 3 | 1 | 1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 1 | 0 | 1 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 |

**Q** Consider the following Graph:



(1) Put arrows on the 8-edges such that the resulting graph is acyclic.



(2) An undirected graph such that such if is impossible to convert is into a "acyclic" directed graph.

∄ any such graph.

Proof: → Given any undirected graph, put a arrow such that it goes from a larger number to a smaller number.

→ As $a > b > c \cdots > a$ is not possible, the resultant directed graph will be acyclic.

# ※ Shortest Path:

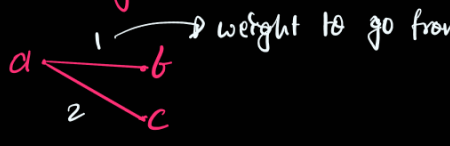
* Single source shorted path (SSSP):

→ Given a directed graph G and a vertex S of G (source), compute shorted path from s to every other vertex of G.

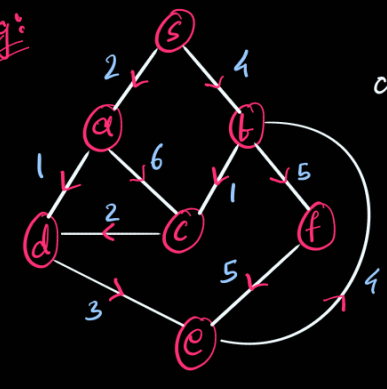
Eg:

→ Run the BFS from s.
The level number for each vertex denotes the distance of that vertex from the source.

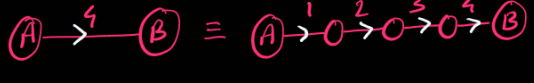
* Weighted edges:




Eg:




dist(s,c) = min(2+6, 4+1)
= 5




→ Modify the graph in the following way, so that we do not need to change the algorithm.

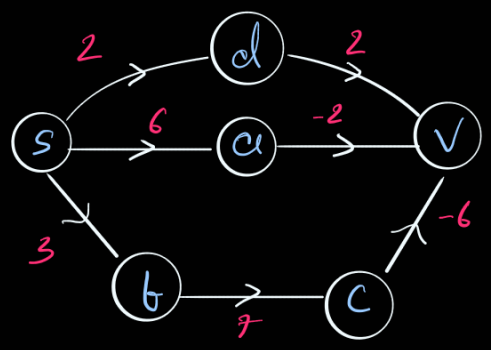
→ This does not work for non-positive integral weights.

# ✱ Bellman - Ford - Moure Algorithm:

```
SSSP(x): BELLMAN - FORD - MOURE ALGO
{
    Set all dist(v) ← ∞ and pred(v) ← null
    while (∃ an edge (u,v) s.t. dist(v) > dist(u) + dist/wt(u,v))
    {
        dist(v) = dist(u) + wt(u,v)
        pred(v) = u
    }
}
```

→ We are not done with just one round because, the updated shortest distance of one node will affect others.

⇒ Let $k(v)$ be the minimum number of edges on a shortest path from $s$ to $v$.




$s \to a \to v : 4$

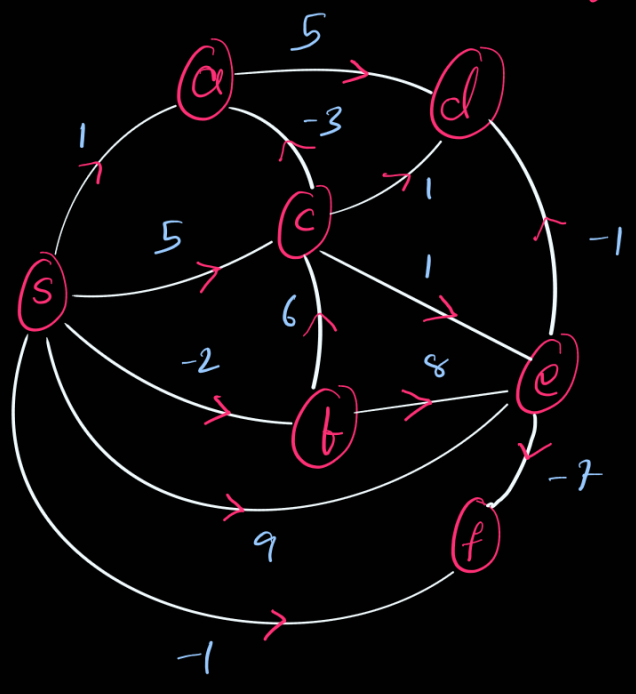
$s \to b \to c \to d : 4$

$s \to d \to v : 4$

Eg:

✓ +0.5

✓ −0.5




| Round | s | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| 0 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 1 | **0** | **1** | **-2** | 5 | ∞ | 9 | -1 |
| 2 | 0 | 1 | -2 | **4** | 6 | 6 | -1 |
| 3 | 0 | 1 | -2 | 4 | 5 | **5** | **-2** |
| 4 | 0 | 1 | -2 | 4 | **4** | 5 | -2 |

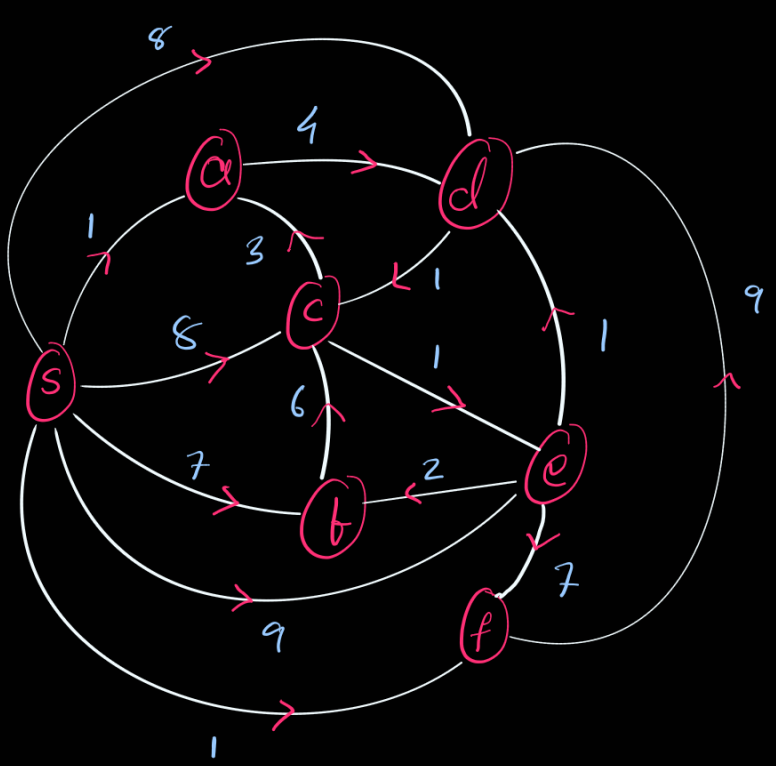
# (n-1) Rounds

- In each round, look at all of the m edges.
- The maximum possible value of the $x(v) = n-1$
- One extra round to check if nothing is getting updated.

$$\text{Running Time} = O(m \cdot n) = O(\underbrace{m}_{\text{each round}}(\underbrace{(n-1)}_{\text{no. of rounds}}+1))$$

---

※ All edges have +ve weightages:




Example of dry run:

In Round 2,

For c,
look at all of the incoming edges to c,

⇒ s, d, b

curr_val
= min( 1st round val + edge )

for s ⇒ 0+8,
d ⇒ 8+1,
b ⇒ 7+6,

= 8

| Round | s | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| 0 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 1 | 0 | 1 | 7 | 8 | 8 | 9 | 1 |
| 2 | 0 | 1 | 7 | 8 | 5 | 9 | 1 |
| 3 | 0 | 1 | 7 | 6 | 5 | 6 | 1 |
| 4 | 0 | 1 | 7 | 6 | 5 | 6 | 1 |

Running Time = $O(m+n\log n)$

NOTE: Doesn't work with -ve weightages.

Soln:

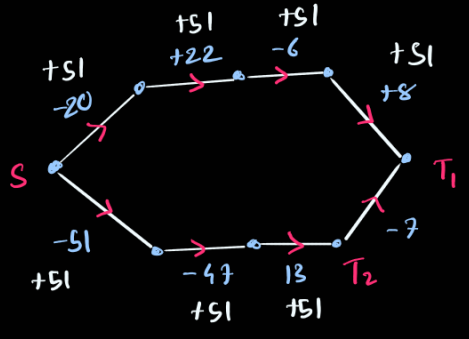
Find the most -ve edge

(suppose $-a$ $(a>0)$)

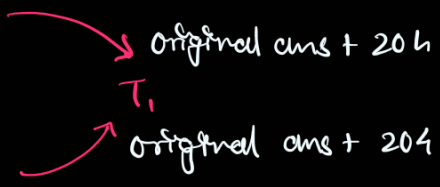
Then, add $a$ to all of the values of $a$.

Q: Construct a graph (without -ve cycles (without negative cycles) with 8 edges having the mentioned original edge weights, such that Dijkstra's algorithm fails to give the correct shortest path, when implemented on the updated edge weights.

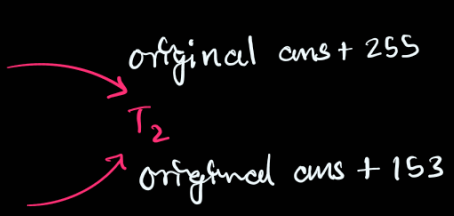
Soln:



For $T_1$ :




For $T_2$ :




} Fails in this case

---

```
DIJKSTRA(s)
{
    for(all vertex v)
        INSERT(v, dist(v)) into Q
    while(Q is non empty){
        U ← ExtractMin from Q
        for(all edges (u,v)){
            if(dist(v) > dist(u) + wt(u,v))
            {
                dist(v) = dist(u) + wt(u,v)
                DecreaseKey(v, dist(v)) in Q.
            }
        }
    }
}
```

DRY RUN :

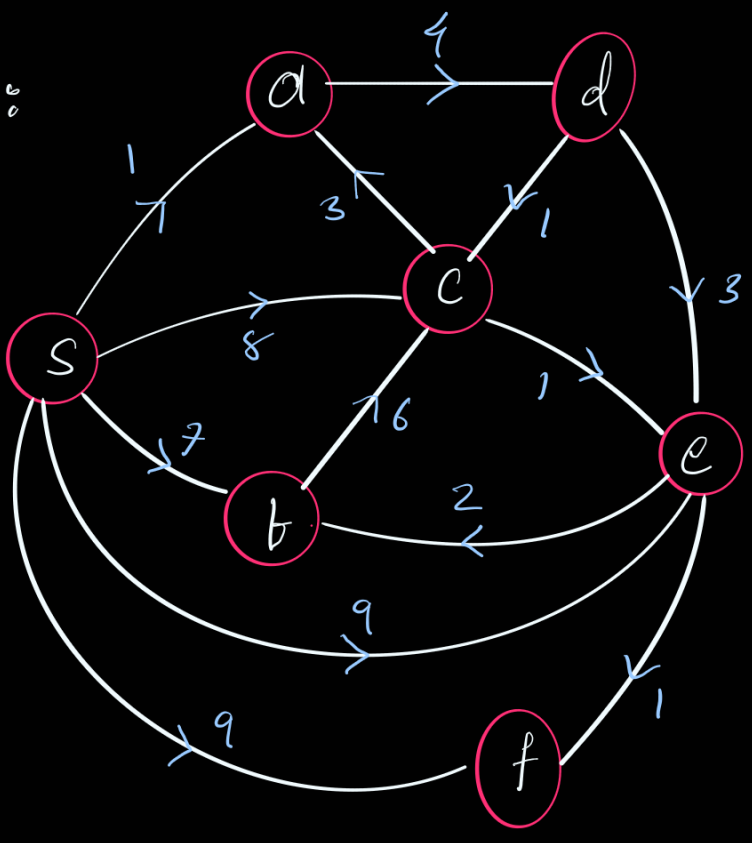
Queue Initially :

(s, 0) (e, ∞)
(a, ∞) (f, ∞)
(b, ∞)
(c, ∞)
(d, ∞)

1) (s, 0)

Queue after step 1:

(a, 1) (e, 9)
(b, 7) (f, 9)
(c, 8)
(d, ∞)




2) (a, 1)

Queue after step 2:

(b, 7) (e, 9)
(c, 8) (f, 9)
(d, 5)

3) (d, 5)

Queue after step 3:

(b, 7) (e, 8)
(c, 6) (f, 9)

4) (c, 6)

(b, 7) (e, 8) (f, 9)

5) (b, 7)

(e, 8) (f, 9)

6) (e, 8)

(f, 9)

7) (f, 9)

Q is empty.

Ans =

| | |
|---|---|
| $s \to 0$ | $e \to 8$ |
| $a \to 1$ | $f \to 9$ |
| $d \to 5$ | $c \to 6$ |
| $b \to 7$ | |

→ Using Fibonacci heaps, all of the mentioned heap operations can be done in $O(\log n)$ time.

Total running time = $O(m + n \log n)$

---

→ Suppose each edge represent the amount of time to travel on that road/street, and each vertex weight represents the amount of time spent at that junction / traffic signal.

Q. How will you modify DIJKSTRA to find a shortest path? Which lines would you add/delete/edit from DIJKSTRA?

Soln:

```
if (dist(v) > dist(u) + wt(u,v))
{
    dist(v) = dist(u) + wt(u,v)
    DecreaseKey(v, dist(v)) in Q.
}
```

```
if ( dist(v) > dist(u) + wt(u,v) + val(u) )
{
    dist(v) = dist(u) + wt(u,v) + val(v)
    Decreasekey (v, dist(v)) in Q.
}
```

---

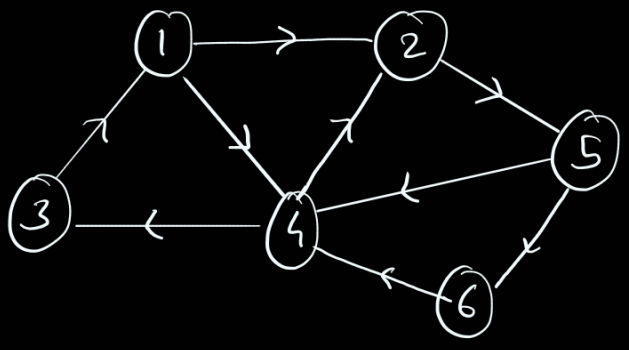
**Q.** You are not allowed to touch the code of DIJKSTRA. Now what will you do?

**Soln:** Update `wt(v,u) += val(u);`

---

**Q.** Draw a directed graph on 6 vertices and 9 edges which has no source or sink vertex.

**Soln:**



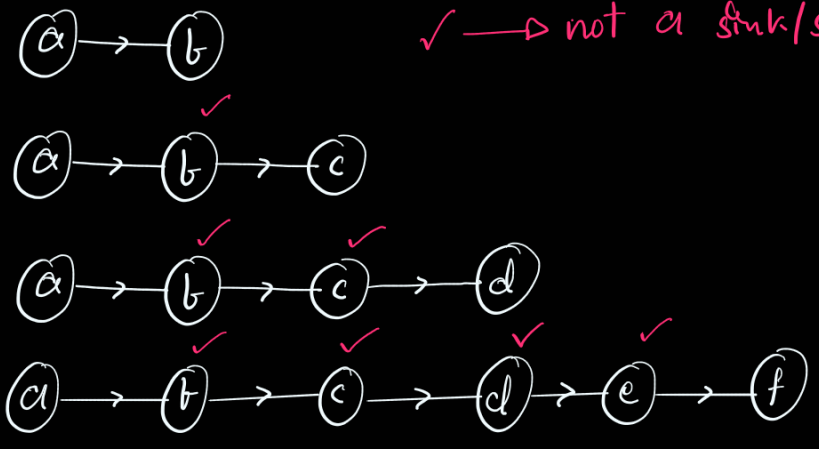

---

Q: Draw a directed acyclic graph on 6 vertices and 9 edges which has no source or sink.

Soln: Impossible.

Proof: By Contradiction.

Let's assume it is possible.




✓ → not a sink/source

Now, $f$ must be an outgoing edge from (f).
& incoming edge to (a).

→ While satisfying any of the above two requirements, it will lead to an acyclic graph.

---

Q: Draw a directed graph on 6 vertices and 5 edges which has no source or sink vertex.

Soln: Impossible.

Every such graph contains atleast one sink/source.

FACT: Every directed acyclic graph has at least one source and atleast one sink.

---

## ※ Topological Sorting:

→ Arrangement of the vertices of a directed acyclic graph on a straight line from left to right, such that the direction of every arrow is from left towards right.

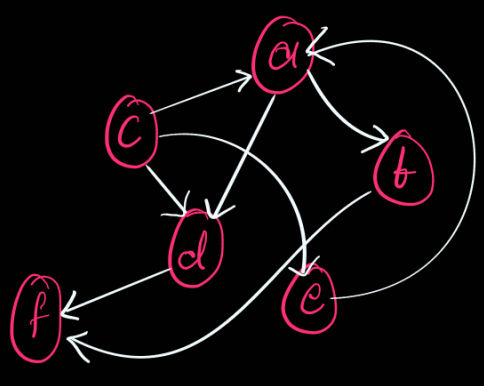
i.e.

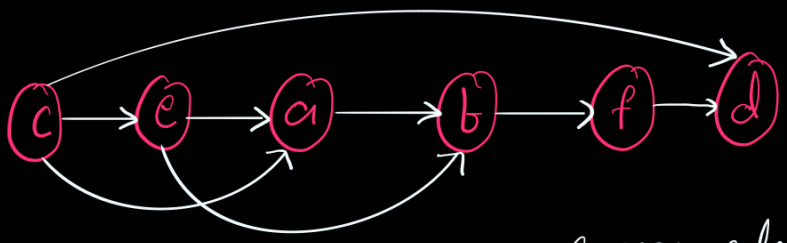
If directed edge $(u, v)$ from vertex $u$ to $v$, $u$ comes before $v$ in the ordering

Fact:

If a graph has a topological sorting, then it is a directed acyclic graph.

Eg:






every edge goes from left to right.

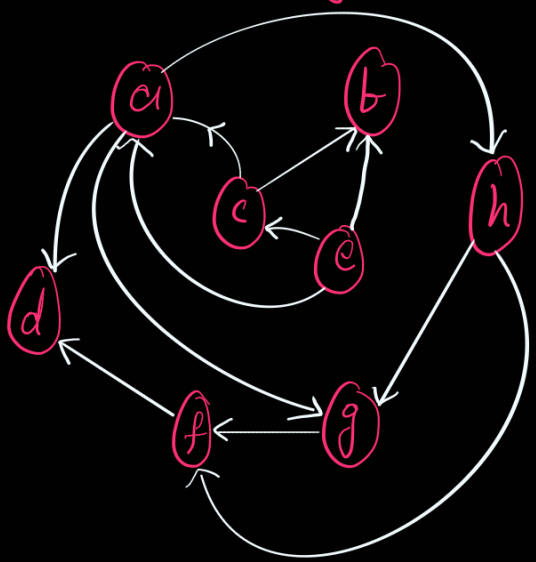
FACT: Given a directed acyclic graph, it is possible to topologically sort it's vertices in $O(n)$ time.

Two more FACTS: ① Every directed acyclic graph (DAG) has atleast 1 source vertex and 1 sink vertex.

② If you remove a vertex (and all the corresponding edges) from a DAG, the remaining graph is a DAG.

---

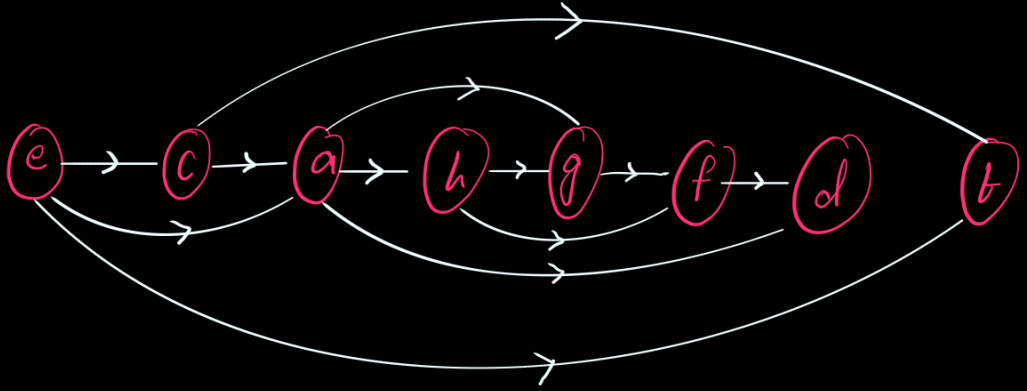
Q: Draw a topological sorting of the given graph:




Hint:

leftmost vertex must be a source & the rightmost vertex must be a sink.

Soln:



⇒ a → d, g, h
b →
c → a, b
d →
e → a, b, c
f → d
g → f
h → f, g

| out-deg | in-deg |
|---|---|
| a : 3 | a : 2 |
| b : 0 | b : 2 |
| c : 2 | c : 1 |
| d : 0 | d : 2 |
| e : 3 | e : 0 |
| f : 1 | f : 2 |
| g : 1 | g : 2 |
| h : 2 | h : 1 |

Time complexity to calculate out-deg & in-deg = $O(|V|+|E|)$

Topological order: (e) (c) (a) (b) (h) (g) (f) (d)

Keep decrementing in-degree;
add when it is zero.

$O(|V|+|E|)$

If graph is not a DAG, then at one step, all of the vertices will have $>0$ in-degree.

Q: Suppose you are given a directed graph $G(V, E)$. How will you check if it is a DAG in $O(|V|+|E|)$ time?

Soln: Do topological sorting & check if in-degree for all vertices is zero at the end.

Q: If YES, output a topological order,
If NO, output a cycle.

Soln: Change the corresponding out-degree too.
Start with vertex whose out-degree $!=0$.

Rather make $adj(G^T)$ $(O(V+E))$
and check for in-degree.

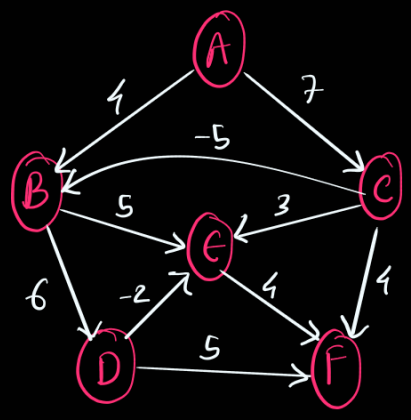
Q: How will you use topological sorting to find shortest path in a DAG in $O(|V|+|E|)$ time.

Soln:

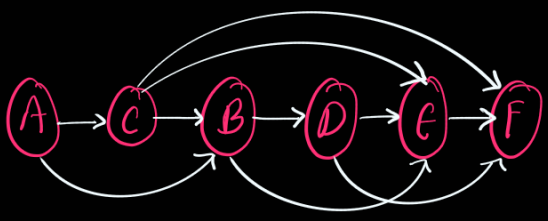
Step ①: Perform Toposort.

Step ②: Traverse in the adjacency list but in the order obtained through toposort.

For example,
consider the following graph:




Topological order:




The order of traversing is known, no need to make any priority queue or etc.

A → B, C
B → D, E
C → B, E, F
D → E, F
E → F
F

At first, take (A),

initialize dist(A) = 0 & all other dist with ∞.

then dist(B) = min(dist(A) + wt(A,B), dist(B))

Similarly, dist(C) = min(dist(A) + wt(A,C), dist(C))

Also, decrease the in-deg of B & C by 1.

Then, go to (C), [next to A in topological order] and traverse its neighbours.

And so on....

Once the in-deg of a node becomes zero, its dist from source node gets fixed.

Time Complexity : $O(|V|+|E|)$

---

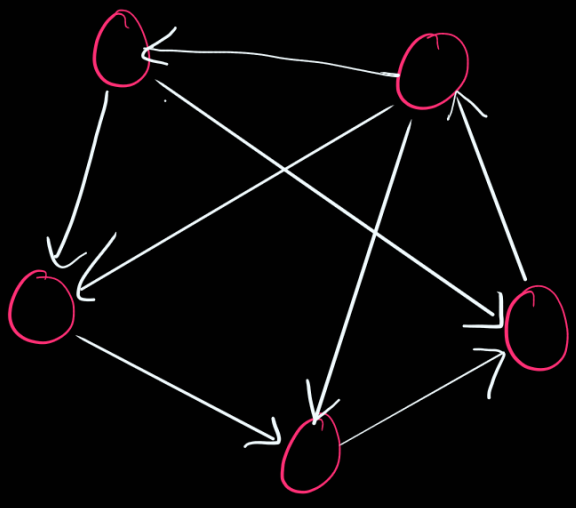
Q. Find the largest simple path / cycle → ∞ in a directed graph.

# ✗ Strongly Connected Components :

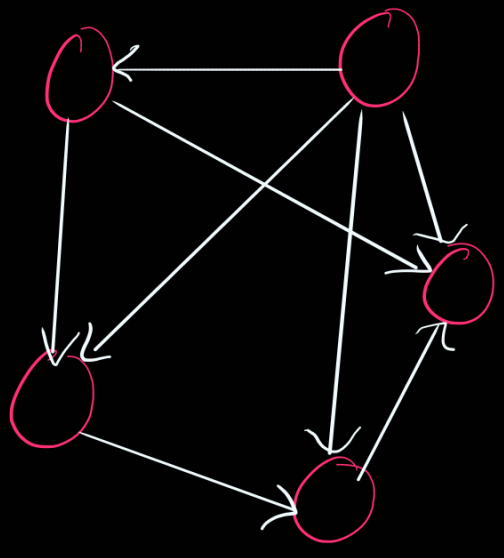
→ Every pair of vertices are reachable from each other.

→ Graph G is strongly connected if, for every u and v in V, there is some path from v to u.

Eg:



Strongly Connected



Not Strongly Connected

# Finding Strongly Connected Components:

**Input:** A directed graph $G = (V, E)$

**Output:** A partition of $V$ into disjoint sets so that each set defines a strongly connected component of $G$.

## Algorithm:

(1) Call DFS($G$) to compute finishing times $f(v)$ for each vertex $v$.

(2) Compute $G^T$

The graph $G^T$ is the transpose of $G$, which is visualized by reversing the arrows on the directed graph.

(3) call DFS($G^T$), but in the main for loop of DFS, consider the vertices in order of decreasing $f(v)$.

④ Output the vertices of each tree in the

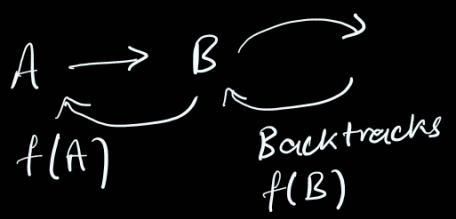
Time Complexity: $\boxed{O(E+V)}$

→ $f(A)$ = finishing time of $V$.
i.e. the time when the DFS algorithm "backtracks" from that node.

→ Consider two node A and B. There is a path from A to B but not from B to A.
Compare their finishing time (compare $f(A)$ & $f(B)$)

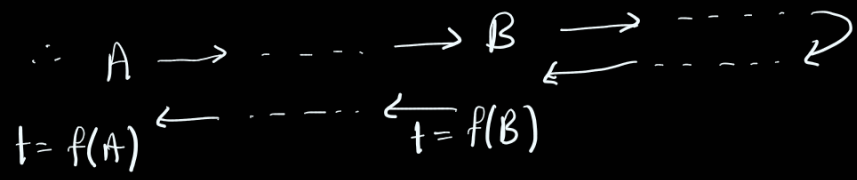
⇒ $\boxed{f(A) > f(B)}$ Ans

Reason: Since DFS starting from A will (recursively) visit B before finishing A.




Or take 2 case:

① B not visited.



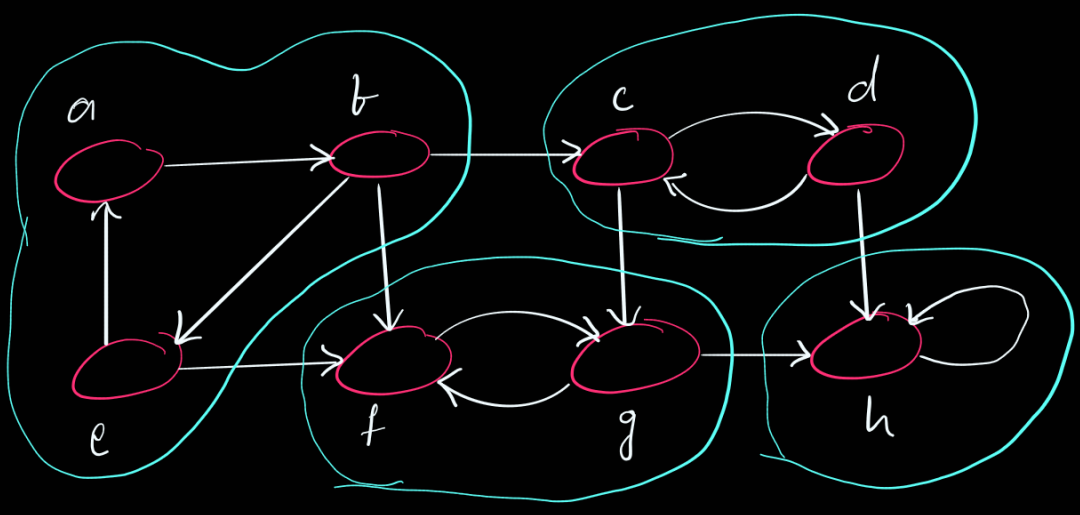

Thus $f(A) > f(B)$

(2) B already visited

∴ clearly, f(A) > f(B)

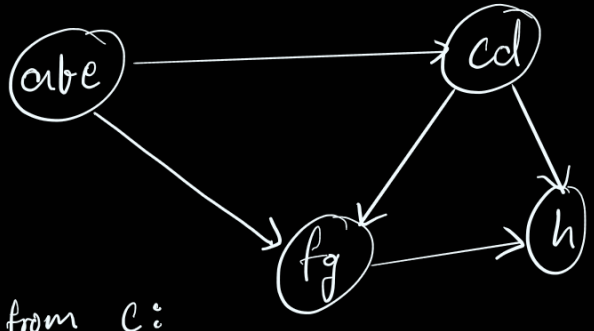
Eg: [Here, a/b means : a → starting time
b → finishing time]

Graph:

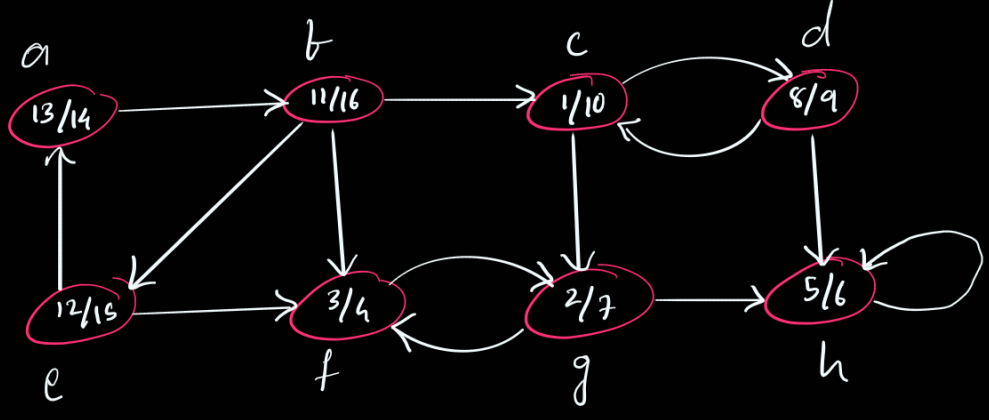



⇓

SCC Graph:




Start DFS from c:



→ Let $C = \{C_1, C_2, \ldots, C_n\}$ be the set of strongly connected components of $G = (V, E)$.

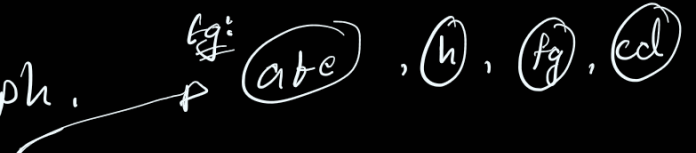
The SCC graph is

$G^{scc} = (V^{scc}, E^{scc})$ where $V^{scc} = \{v_1, \ldots, v_n\}$ where each $v_i$ in $V^{scc}$ represents the strongly connected component $C_i$ in $C$.

There is a directed edge $(v_i, v_k)$ in $E^{scc}$ if there is a directed edge $(x, y)$ in $E$ s.t. $x \in C_i$ and $y \in C_k$.

**Why does it work:**

Consider the above SCC graph.




The vertices inside a node of the SCC graph can be reached even if the edges of the original graph are reversed starting from any vertex among the vertices of a particular SCC.

Since DFS is performed on the node with highest time (finishing) in the graph with reversed edges, it can be observed that the DFS tree will be limited to one strongly connected component.

Observe that in the above SCC graph,

$$f(abc) > f(cd) > f(fg) > f(h)$$

→ The dfs will reach all nodes within the SCC due to cycles that are unaffected by reversed edges.

However, the DFS will not be able to leave the scc due to the reversed edges across components.

Questions:

1] $((G^T)^{scc})^T = G^{scc}$ (True/False)

2]

- Consider a directed graph G=(V,E) where each node is initially colored white. What should be the minimum number of nodes that we should change to red such that for each white node *v* in G, there exists at least one red node *r* such that there is a directed path from *r* to *v* and *v* to *r*. Assume that for every node there is at least one other node it is reachable from and can reach to.

Soln: 1) TRUE

2) no. of SCCs

More Questions:

1. Consider a party where you would want to group people who should sit together at the same table such that at each table among any pair of people, one is known to the other either directly or through some sequence of people on that table. We would want to minimise the number of tables. Give an algorithm for each of the below scenarios

Case1: If guest g knows guest t then guest t knows guest g.

Case2: If guest g is known to guest t, it is not necessary that guest t be known to guest g.

2. Assume every pair of guest knows each other directly. Two guests who are on bad terms are not to be seated on the same table. What are the minimum number of tables required.

Solution:

1)

Case (1): Find no. of disconnected graphs using DFS.

Case (2): Use the above given algo to find no. of SCCs.

2) even cycle => 2
   odd cycle => 3

} Think about it.

---

Additional Question:

Find the largest simple path / cycle $\rightarrow \infty$ in a directed graph.