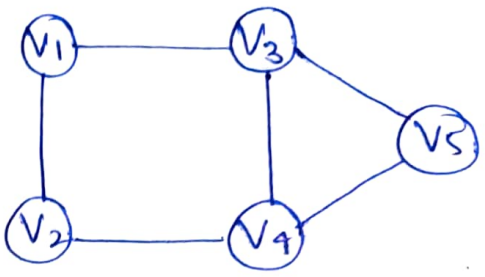
GRAPH DATA STRUCTURE

$G = (V, E)$

$V = \{V_1, V_2, V_3, V_4, V_5\}$

$E = \{(V_1, V_2), (V_1, V_3), (V_2, V_4), (V_3, V_4), (V_3, V_5)\}$



Directed - Edges have direction

$|V| * (|V|-1)$

(V_1, V_2) → we can go to V_2 from V_1 but not vice-versa

Undirected - (V_1, V_2) → we can go backwards also.

max edges

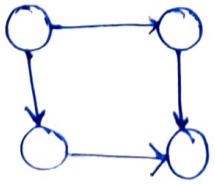
→ $\frac{|V| * (|V|-1)}{2}$.

Walk - sequence of vertices that we get by following edges.

Path - we cannot have repeated vertices

Cyclic graph - If there exist a walk on the graph that begins and ends with same vertex.

DAG Directed acyclic graph



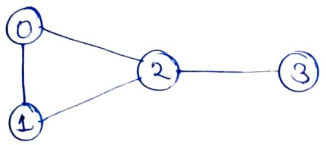
- No cycle
- directed

weighted graph

Edges are assigned weights..

Graph Representation

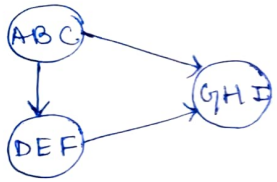
(a) Adjacency Matrix



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

Always a symmetric matrix in case of undirected graph.

- If we have vertex names as strings we can use hashmap for storing them & assigning an 'int' for them



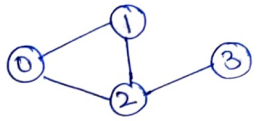
	0	1	2
0	0	1	1
1	0	0	0
2	0	1	0

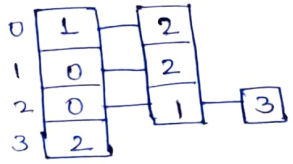
ABC	0
GHI	1
DEF	2

- space required - $\Theta(V \times V)$
- check if u and v are adjacent - $\Theta(1)$
- Find all vertices adjacent to u - $\Theta(V)$
- Find degree of a vertex - $\Theta(V)$
- Adding and removing an edge - $\Theta(1)$
- Adding and removing a vertex - $\Theta(V^2)$

Adjacency List

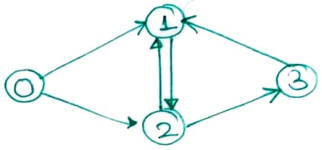
- only store vertices which are adjacent to a vertex

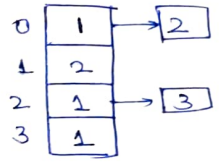




Adjacency list

- we have an array of list
  - → either be linked list
  - → dynamic vector





properties

Space - $\Theta(V+E)$
- undirected $V + 2*E$
- directed $V + E$

- check if there is an edge from u to v - $\Theta(V)$
- Find all adjacent of u. - $\Theta(\text{degree}(u))$
- Find degree of u - $\Theta(1)$
- Adding an edge - $\Theta(1)$
- Removing an edge - $\Theta(V)$.

Implementation :-

- create an array of vectors.

V → no of vertices

```
vector<int> adj[V]

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

Comparison b/w representations

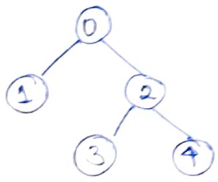
	List	Matrix
Memory	$\theta(V+E)$	$\theta(V \times V)$
Check if there is a vertex from u to v	$\theta(V)$	$\theta(1)$
Find all adj to u.	$\theta(\text{degree}(U))$	$\theta(V)$
Add an Edge	$\theta(1)$	$\theta(1)$
Remove an edge	$O(V)$	$\theta(1)$.

List is better than matrix

Bredth First Search

1st Version - Given an undirected graph and a source vertex 's' print BFS from the given source.

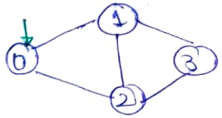
I/P →



s = 0

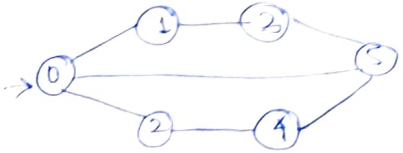
O/P → 0 1 2 3 4

I/P →



O/P → 0 1 2 3

I/P →



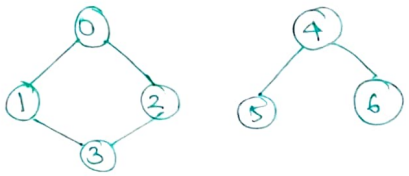
O/P → 1 2 5 3 4

- For ensuring that an item should be processed only once, we will maintain a boolean array.
- The approach will be same as Level order traversal of a tree

```
void BFS (vector<int> adj[], int V, int s) {
    bool visited[V+1];
    for (int i=0; i<V; i++) {
        visited[i] = false;
    }
    queue<int> q;
    visited[s] = true;
    q.push(s);
    while (q.empty() == false) {
        int u = q.front();
        q.pop();
        cout << u << " ";          <-- Printing curr node
        for (int v : adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}
```

2nd Version - source is not given and graph may be disconnected

```
void BFS_D (vector<int> adj[], int V) {
    bool visited[V+1];
    for (int i=0; i<V; i++)
        if (visited[i] == false) BFS (adj, i, visited);
}
```



⇒ 0 1 2 3 4 5 6

time complexity → O(V+E)

To find number of connected components

We can basically add a variable named count in the main loop.

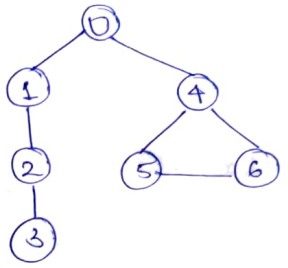
```
for (int i=0; i<V; i++){
    if (visited[i] == false){
        BFS(adj, i, visited);
        count++;
    }
}
return count;
```

Applications of BFS

- Shortest path in an unweighted graphs
- Crawlers in search Engines
- In Garbage collections (Cheney's Algorithm)
- Cycle Detection
- Ford Fulkerson Algo
- Broadcasting

Depth First search

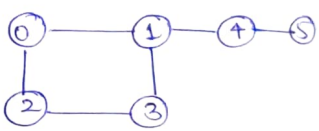
I/P →



S=0

O/P → 0 1 2 3 4 5 6

I/P :



S = 0

O/P =

0 1 4 5

3 2

```
void DFSRec(vector<int> adj[], int s, bool v[])
    visited[i] = true;
    cout << s << ' ';
    for (int u : adj[s])
        if (visited[u] == false)
            DFSRec(adj, u, visited);
}
⇒ void DFS(vector<int> adj[], int V, int s) {
    bool visited[V];
    for (int i=0; i<V; i++)
        if (visited[i] = false)
    DFSRec(adj, s, visited);
}
```

- Disconnected graph

```
for (int i=0; i<V; i++) {
    if (visited[i] == false)
        DFSR(adj, i, visited);
}.
```

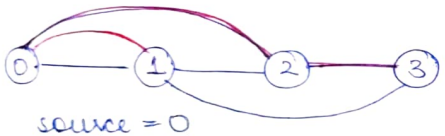
Application of DFS

1) Cycle detection
2) Topological sorting
3) Strongly connected components
4) Solving Maze and similar puzzles
5) path finding

Shortest Path in unweighted Graph

↳ As there is no weights, the ~~mini~~ shortest path is the one having minimum edges in between.

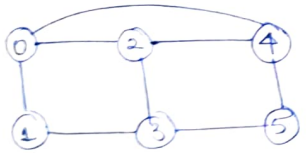
I/P →



source = 0

O/P → 0 1 1 2

I/P →



source = 0

O/P → 1 1 2 1 2

- We can use BFS here and we can maintain a distance vector where while traversal we can update the distance as

dist[v] = dist[u] + 1

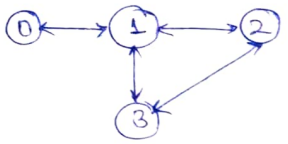
1) Initialize dist[V] = { INF, INF, ... ∞ }
2) dist[s] = 0
3) create a queue.
4) ~~tr~~ Initialize: visited[V] = { false, false ... }

q.push(s), visited[s] = true.

```
while (q is not empty) {
    u = q.pop();
    for every adjacent v of u {
        if ( visited[v] == false) {
            dist[v] = dist[u] + 1   <- update in dist
            visited[v] = true
            q.push(v);
        }
    }
}
```

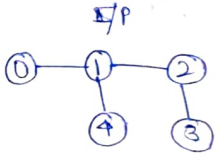
Detect cycle in undirected Graph

I/P



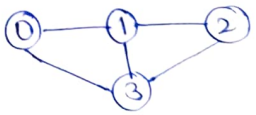
O/P → Yes.

I/P



O/P → NO.

I/P



O/P → Yes.

We can use both DFS and BFS in this, but the corner cases is, we ~~can~~ assume.

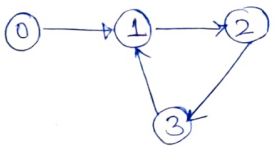
(1) approach is if we see a visited vertex we say there is a cycle but the vertex may be the parent itself

for 2, 1 is visited by 1 is the parent

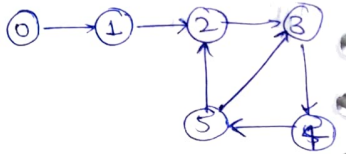
Hence we pass an extra parameter in the DFS call and check if the current vertex is parent itself

```
bool DFSRec(adj[], parent, visited, s) {
    visited[s] = true
    for (int v : adj) {
        if (visited[v] == false) {
            x = dfsRec(adj, parent, visited, v)
            if (x) return true
        }
        else if (v != parent)
            return true;
    }
    return false;
}
```

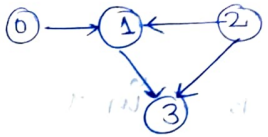
Directed type — Detect cycle in Directed graph

I/P → 

O/P → Yes.

I/P : 

O/P → Yes

I/P → 

O/P → NO.

DFS(0)
→ DFS(1)
DFS(2)
→ DFS(3)
→ DFS(4)
→ DFS(5)

back edge

There is an edge from one of the descendants to ancestor

Loop ←

- We have to maintain the list of ancestors.

For this we will maintain an array named rec[]

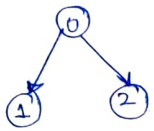
```
DFSRec(adj, s, visited[], rec[]) {
    visited[s] = true
    rec[s] = true
    for (auto v : adj[s]) {
        if (visited[v] == false &&
            DFSRec(adj, s, visited, rec))
            return true;
        else if (rec[v] == true) return true
    }
    rec[v] = false;
    return false;
}

DFS(adj, V) {
    visited[V] = {false}, rec[V] = {false};
    for (int i=0; i<V; i++) {
        if (visited[i] == false)
            if (DFS(adj, i, visited, rec)) return true;
    }
    return false;
}
```

TOPOLOGICAL SORTING (Kahn's Algo)

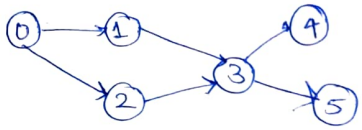
- for acyclic graph only

I/P →



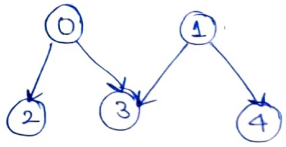
O/P → 0 1 2

I/P →



0 1 2 3 4 5

I/P →

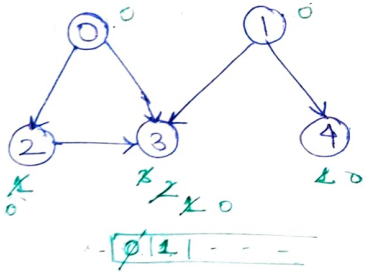


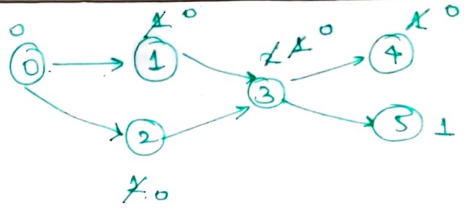
0 1 2 3 4 / 0 1 3 2 4

If there is edge from 0 → 1 then 1 must be printed only after 0

(Kahn's Algo) BFS Based solution

- Store indegree of each vertex
- create queue 'q'
- add all the vertices with indegree 0
- while (q is not empty) {
  1) int u = q.pop();
  2) print u;
  3) For every adjacent v of u
     1) Reduce the indegree by 1
     2) If indegree of that vertex becomes 0 push that to queue.





queue

O/P → 0 1 2 4 5

Implementation

- create an array indegree[V] = {0}.
  - → If I have control over addEdge function I will increase the indegree of v when there is a directed edge from u → v indegree[v]++;
  - → Else, traverse the graph (adj) and increament the indegree

time comp → O(V+E)

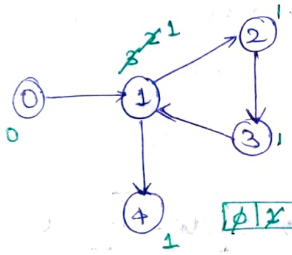
Detect cycle in Directed graph using Kahn's Algo

- If there is a cycle then at a point there will not be any vertex with indegree 0, all are dependent on each other
- we'll use this concept for finding cycle
- we'll maintain a variable named count will will keep track of all the vertices processed by the topo. algo.
- If count < V, it clearly states there is a cycle.

Algorithm (modified)

1) store indegree
2) create a queue q.
3) add all 0 indegree vertices to the q.
4) count = 0
5) while (q is not empty) {.
   (a) u = q.pop();
   (b) for every adjacent v of u
      1) Reduce indegree[v] by 1
      2) If (indegree[v] == 0) push v to q.
   (c) count ++;
}
return (count != V)

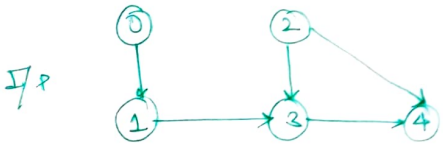
time comp → O(V+E)



ϕ	x	=

O/P → 0, 1.
bt count = 2
2 ≠ 5 true
cycle

Topological sorting using DFS

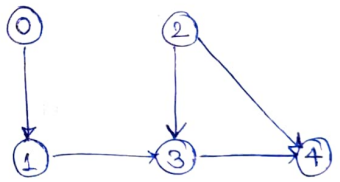


O/P → 0 1 2 3 4

- If we'll create a stack s and push a ~~vertex~~ vertex when we've processed it completely. and all its descendants have already pushed.

1) create an empty stack st
2) For every vertex u, do following
   if (u is not visited)
      DFS (u, st)
3) while (st is not empty)
   pop an element & print it.

```
DFS(u, st){
  1) Mark u as visited
  2) For every adjacent v of u
       if (v is not visited)
          DFS (v, st)
  3) push u to st.
```



stack

4	3	1	0	2

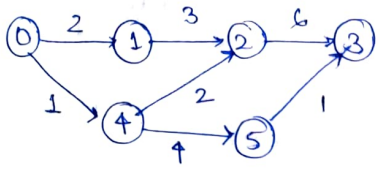
~~stack~~

~~0~~	~~1~~	3	~~4~~

DFS (0)
→ DFS (1)
  → DFS (3)
    → DFS (4)
      → st.push (4)
    → st.push (3)
  → st.push (1)
→ DFS st.push (0)
DFS (2)

Shortest Path in DAG

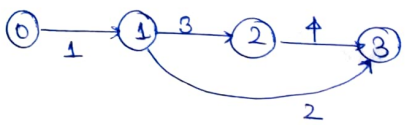
I/P



source - 0

O/P → 0 2 3 6 1 5

I/P :



source = 1

O/P → INF 0 3 2

- We will use topological sort here.

shortestPath (adj, s)

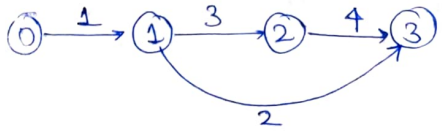
1) Initialize dist[V] = {INF, INF ---- }.
2) dist[S] = 0;
3) Find a topological sort of the graph
4) For every vertex in the topological sort
   a) For every adjacent v of u

Relaxing a vertex: { if (dist[V] > dist[u] + weight(u,v))
dist[V] = dist[u] + weight(u,v))

sol:

dist[6] = {0, INF, INF, INF, INF, INF}.
(0, 1, 2, 3, 4, 5)
(2, 4, 3, 6, 1, 5)

op. sort → 0 1 4 2 5 3



source = 1

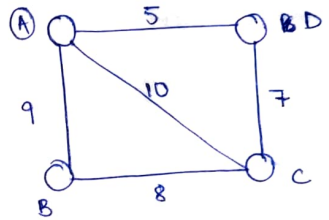
dist[V] = {INF, 0, INF, INF} → INF, 0, 3, 2
(3, 2)

top.s → 0 1 2 3

time comp → θ(V+E)

The purpose of topological sort is to go ~~from~~ in forward direction only.

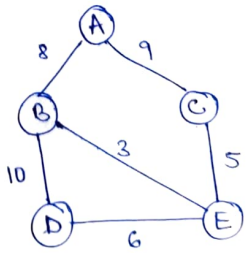
Minimum Spanning Tree



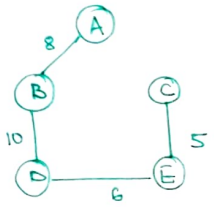
given a weighted and connected undirected graph

Spanning Tree → a subset of graph G having same number of vertices and (V-1) edges minimum edges.

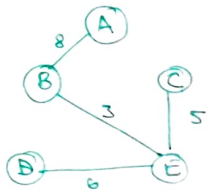
Minimum spanning tree → having minimum weights among all spanning tree (min. total weights)



→

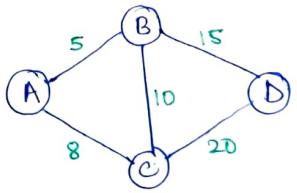


= 29

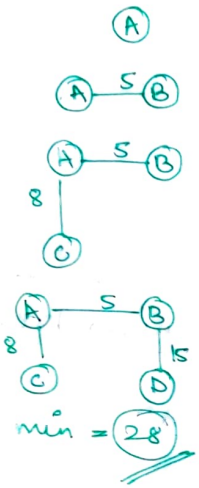


= 22

- It is a greedy algorithm; At any vertex we will find the minimum path to the vertex which is not yet included



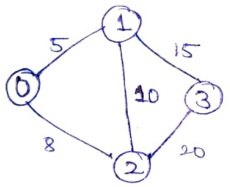
In MST | Not in MST



since the two sets have to be connected we'll pick in the min weighted edge to connect the next element.

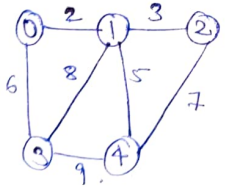
Implementation

I/P : graph[][] = $\begin{bmatrix} 0 & 5 & 8 & 0 \\ 5 & 0 & 10 & 0 \\ 8 & 10 & 0 & 20 \\ 0 & 15 & 20 & 0 \end{bmatrix}$



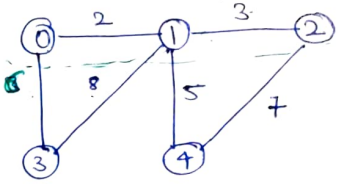
O/P → 28

I/P :- graph[][] = $\begin{matrix} & 0 & 1 & 2 & 3 & 4 \\ 0 & \\ 1 & \\ 2 & \\ 3 & \\ 4 & \end{matrix}\begin{bmatrix} 0 & 2 & 0 & 6 & 0 \\ 2 & 0 & 3 & 8 & 5 \\ 0 & 3 & 0 & 0 & 7 \\ 6 & 8 & 0 & 0 & 9 \\ 0 & 5 & 7 & 9 & 0 \end{bmatrix}$



O/P → 16

Initially : res = ~~0~~ 2 ~~5~~ ~~10~~ 16



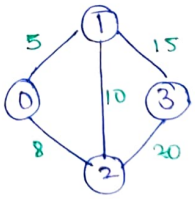
MST Set = {0}.
→ {0, 1}
→ {0, 1, 2}
→ {0, 1, 2, 4}
{0, 1, 2, 4, 3}

~~int~~ ~~print~~MST (~~vector<int>~~ graph[], ~~int~~ V){

① intialise a mset[] → {F, F, F, F} ← no edge is included

② ~~dist~~ create an array named key initially ∞
key[V] = {0, ∞, ∞, ∞}
↑ will always start from 0

③ find the minimum of array key and then update the ~~addr~~ distance of its adjacent nodes.



Initially key[] → {0, ∞, ∞, ∞}
mset[] → {F, F, F, F}

count = 0 u = 0
mset[] → {T, F, F, F}
Key[] → {0, 5, 8, ∞}.

count = 1 u = 1
mset[] → {T, T, F, F}
Key[] → {0, 5, 8, 13}

count = 2, u = 2
mset[] → {T, T, T, F}
key[] → {0, 5, 8, 15}.

count = 3 u = 3
mset[] → {T, T, T, T}
~~Key → {0, 5, 8, 15}~~
res = 28

key[i] ← minimum edge choosen to get the i from parent.

```
int primeMST ( vector < int > graph [], int V ) {
    int key[V], res = 0
    fill ( key, key + V, INT_MAX );
    key [0] = 0;
    bool mset [V] = { false };
    for ( int count = 0 ; count < V ; count++ ){
        int u = - 1 ;
        for (int i = 0 ; i < V ; i++ ){
            if ( !mset[i] && (u == -1 || key[i] < key[u])
                u = i ;
        }
        mset [u] = true;
        res = res + key [u];
        for (int vv = 0 ; vv < V ; vv++) {
            if ( graph [u][vv] != 0 && !mset [vv])
                key [vv] = min ( key [vv], graph [u][vv]);
        }
    }
    return res
}
```

getting next min key

updating distance of neighbouring vertices

time complexity → $O(V^2)$

can be optimised by using
→ min heap DS for storing keys
→ adjacency list

$O((V+E)\log V)$

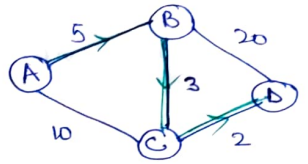
$O(V \log V)$

Dijkstra's Algorithm

Problem:- given a weighted graph and a source find shortest distances from source to all other vertices.

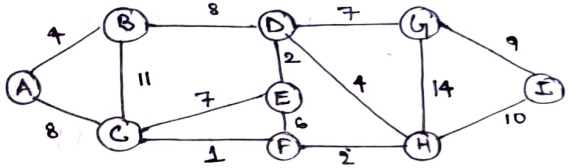
- directed or undirected

I/P →



source = 10

A — 0
B — 5
C — 8
D — 10



O/P

A — 0
B — 4
C — 8
D — 12
E — 14
F — 9
G — 19
H — 11
I — 21

- initialize the distance vector as

A	B	C	D	E	F	G	H	I
0	∞	∞	∞	∞	∞	∞	∞	∞
0	4	8	∞	∞	∞	∞	∞	∞
0	4	8	12	∞	∞	∞	∞	∞
0	4	8	12	15	9	∞	∞	∞
0	4	8	12	14	9	19	16	∞
0	4	8	12	14	9	19	16	∞
0	4	8	12	14	9	19	16	21
0	4	8	12	14	9	19	11	21
0	4	8	12	14	9	19	11	21
0	4	8	12	14	9	19	11	21

- go to adjacent to current node and relax

```
Relax (u, v){
if (d[v] > d[u] + w(u,v))
    d[v] = d[u] + w(u, w);
}
```

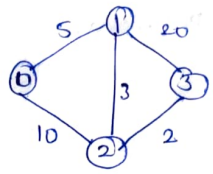
pick min from dist. vector and finalize it and look for its adjacents

- does not work for negatively weighted edges

Implementation

I/P → graph[][] = $\begin{bmatrix} 0 & 5 & 10 & 0 \\ 5 & 0 & 3 & 20 \\ 10 & 3 & 0 & 2 \\ 0 & 30 & 2 & 0 \end{bmatrix}$

src = 0




```
vector<int> dijkstra (vector<int> graph[],
                      int V, int src){

    vector<int> dist (V, INT_MAX);
    dist[src] = 0
    vector<bool> fin (V, false);
    for (int cnt=0 ; cnt< V-1 ; cnt++) {.
```

finding min.

```
        int u = -1
        for (int i=0 ; i<V ; i++){
            if ( !fin[i] && (u == -1 ||
                 dist[u] > dist[i])
                u = i ;
        fin[u] = true;
```

Relaxing adjacents

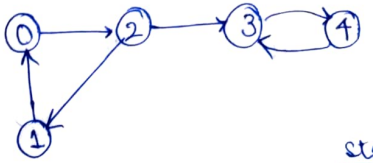
```
        for (int v=0 ; v<V ; v++){
            if (graph[u][v] != 0 &&
                fin[v] == false ){
                dist[v] = min( dist[v],
                          dist[u] + graph[u][v]);
    }.
    return dist;
}.
```

optimised approach

- using priorityqueue (min-heap)

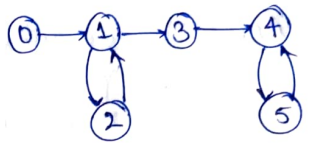
  → O(V+E) log V

Kosaraju's Algorithm

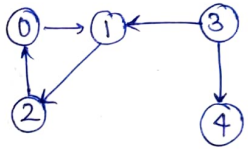
I/P → 

O/P → 0 1 2
3 4

strongly connected components – the vertices are connected in a way so that they ~~the~~ are reachable from each other.

I/P → 

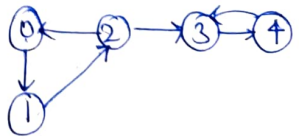
O/P → 0
1 2
3
4 5

I/P : 

O/P → 0 1 2
3
4

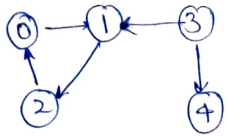
If we find strongly connected components on undirected graph then all those ~~co~~ vertices which are ~~et~~ either connected will be included in one pass. because if u is reachable from v, v is also reachable from u.

⊙ But in directed graph, if we start from 0, we can reach every vertex of the graph but if we start from 4, we can have every strongly connected comp.



4 3
2 0 1

the concept is going backward.



here the main source vertex is 3

we will start from 0

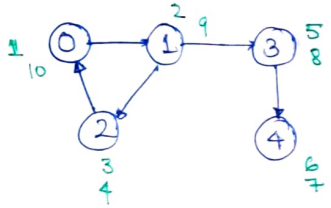
0 1 2
3
4

we want that in (u) → (v)

- we want to process v first
- then u.
  (because u is source)

steps

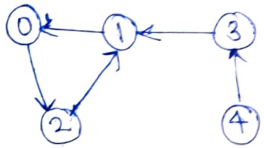
① Order all the vertices in order of their finish times.



0 1 3 4 2

you can start at any vertex

② invert all the edges.



③ DO DFS in the reverted order.

0 2 1
3
4

IMPLEMENTATION OF STEP 1

① Create an empty stack, st

② For every vertex u, do
  if (u is not visited)
    DFSRec (u, st).

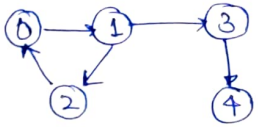
③ while (st is not empty)
  print & pop.

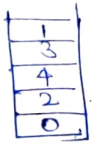
DFSRec (u, st)

(a) Mark u as visited

(b) For every adjacent of u.
  if (v is not visited)
    DFS (v, st)

(c) push u to the stack

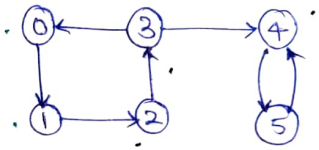


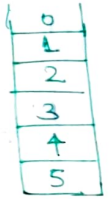


Begin from 1

1 3 4 2 0

Ex-

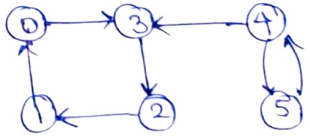




Begin with 0

order → 0 1 2 3 4 5

step-2



o/p → 0 3 2 1
4 5

O(V+E)

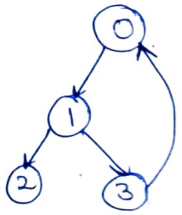
Why we need to reverse the graph?

We can simply move from right to left in the order we got from step 1, but we need to do

Step 2, because we need to make sure that the strongly connected components remains are included

* If we reverse a graph, the strongly connected components remains connected.

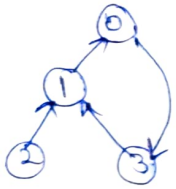
* we have a counter example that if we try to move from right to left and include DFS then we'll get false strongly connected pairs



Step-1 → 0 1 2 3

if we start from 3 we will get all in 1 component 3 0 1 2 X

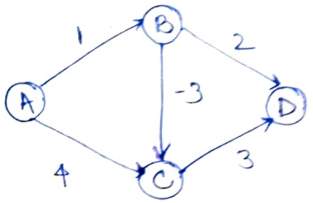
Step-2



0 3 1
2

BELLMAN FORD Algorithm

I/P :



source → A

O/P → {0, 1, -2, 1}

idea → find shortest path for one edge length, then two edge length, then three edge lengths and so on

Algorithm → we relax all edges V-1 times

Algo :- d[V] = {∞, ∞, ∞, ---}.

d[s] = 0

for (count = 0; count < (V-1); count++) {

for (every edge (u,w)

if (d[v] > d[u] + weight(u,v))

d[v] = d[u] + weight(u,v);

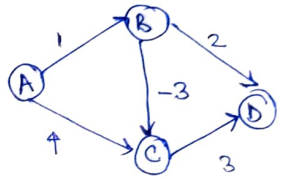
O(VE)

E → V(V-1)

O(V²)

O(V³)

I/P →



As order of edges is not mention we will take any random order

B → C
B → D
C → D
A → B
A → C

O/P distances[] = {0, 1, -2, 1}.

1st iteration

A	B	C	D
0	∞	∞	∞
0	1	~~∞~~ 4	∞

we'll have shortest path for length 1

2nd iteration

A	B	C	D
0	1	~~4~~ -2	~~∞~~ 1

3rd iteration

A	B	C	D
0	1	-2	1
↓			

(no change)

* we don't have shortest path of edge length 3.

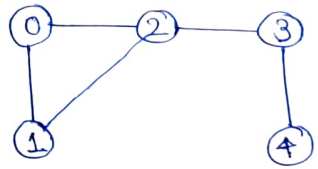
How does it handle negative weighted cycles

- After V-1 iteration if we still get d(v) > d(u) + weight(u,v)

↳ negative cycle encountered

Articulation point

If by removing a vertex and all its associated edges the number of connected components increases by 1 then the vertex is called as articulation point



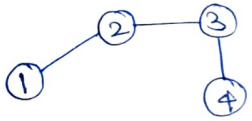
O/P
2,3

connected comp → 1

by removing (2)

connected comp → 2
(2 is articulation point)

Removing (1)



CC = 1
(so 0 is not an articulation point).

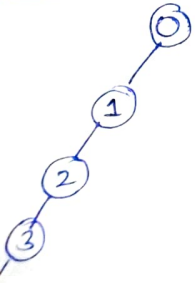
Removing 3

CC = 2
So (3) is also an articulation point

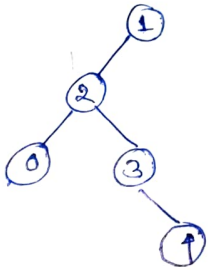
(*) used to find vulnerable points in the network.

We will draw the DFS tree and those vertices whose DFS tree has two children at root will be articulation point.

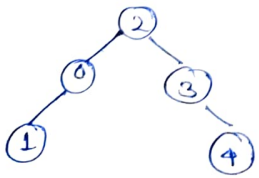
DFS at 0



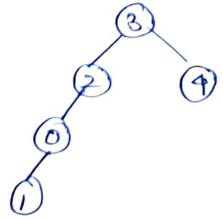
DFS at 1



DFS at 2



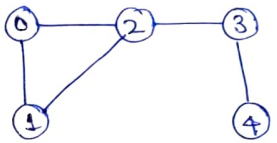
DFS at 3

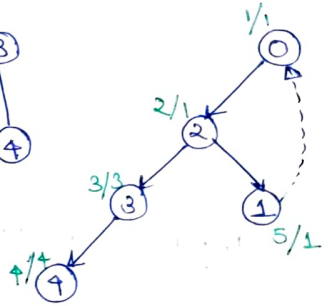


- If a node is making two components connected then only it'll have two children otherwise it'll have only one child as all the other nodes will be accessible through single loop.

Discovery time
time assigned while doing DFS traversal

Low values
↳ lowest discovery time node reachable through that node (via tree or back edges)

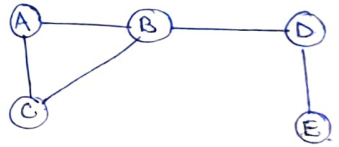




If there is an edge $u \to v$ in DFS tree and low[v] ≥ disc[u] Then u is articulation point

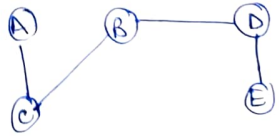
Bridges in the graph

similar to articulation point. In this an 'edge' is called bridge if after removing that edge the number of connected components inc.

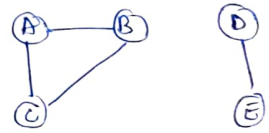


O/P
= BD, DE

(1) Remove AB



(2) Remove BD

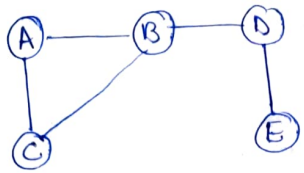


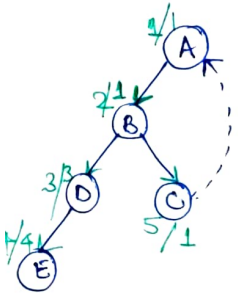
BD is bridge

$u \rightarrow v$

low[v] > disc[u]

then $u \rightarrow v$ is a bridge.





AB → low(B) > disc(A)

1 > 1 ✗

BD → low(D) > disc(B)

3 > 2 ✓

DE → low(E) > disc(D)

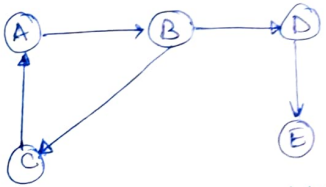
→ 4 > 3 ✓

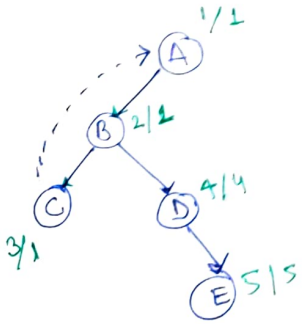
Tarjan's Algorithm

For finding strongly connected components

→ Based on discovery time and low value

disc[u] = low[u]



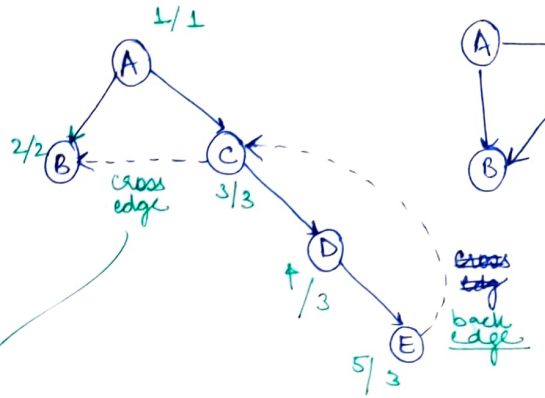


E (pop as disc(E) = low(E))

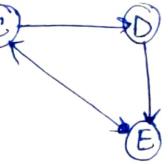
E

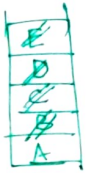
D

C B A



an edge is called cross edge when the node it is going to point is not in function call stack and is already visited

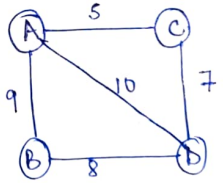


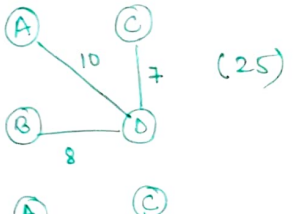


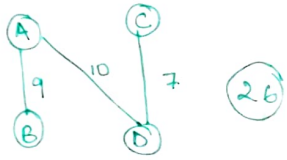
B
E D C
A

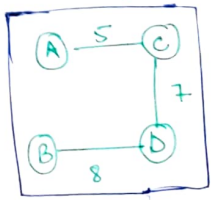
KRUSKAL'S ALGORITHM

For finding MST









Algorithm

① sort all edges in increasing order

② Initialize MST = { }, res = 0

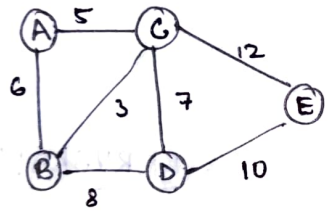
③ Do following for every edge 'e', while MST size does not become V-1.

(a) If adding e to MST does not cause a cycle

MST = MST ∪ {e}

res = e.weight

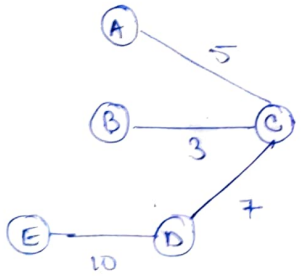
④ return res.



sorted edges

B C	3
A C	5
A B	6 X
C D	7
B D	8 X
D E	10
C E	12

we will stop as V-1 ~~vertex~~ edges are added



~~we~~

For easier implementation will store graph as array of edges.

```
Struct Edge {
    int src, dest, wt;
    Edge (int s, int d, int w) {
        src = s;
        dest = d;
        wt = w;
    }
};
bool myComp (Edge e1, Edge e2) {
    return e1.wt < e2.wt;
}
```

```
int parent[V], rank[V]
int kruskal (Edge arr[]) {
    sort (arr);
    for (int i=0; i<V; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
    for (int i=0, s=0; s<V-1; i++) {
        Edge e = arr[i]
        int x = find (e.src);
        int y = find (e.dst);
        if (x != y) {
            res += e.wt;
            union (x,y);
            s++;
        }
    }
    return res;
}
```

union comp

O(E log E)
+
O(V)
+
O(E log V)
↓
O(E log E)