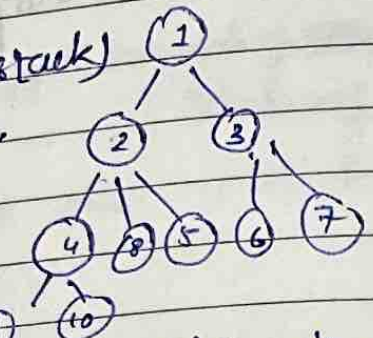
Time complexity of DFS = $O(|V|+|E|)$

Date: __/__/____

BREADTH FIRST SEARCH

→ Thus, we will use a queue because recursion (stack) will give us a wrong order.



Req. Ans: 1 2 3 4 8 5 6 7 9 10

× Ans by recur: 1 2 3 4 8 5 9 10 6 7

✓ Ans by using Q: 1 2 3 4 8 5 6 7 9 10

Recursion is sort of stack

⟹ BFS. Q: 1 ; Q: 2,3 ; Q: 4, 8, 5, 6, 7 . Q: 8, 5, 6, 7, 9, 10

Usually we differenciate between elements i'e, whether they are in the queue, visited, left to visit.

Thus, our flag will typically have three attributes.

pseudo code:-

```
Q = Create Queue (n)
EnQueue (Q, s)
while (! Empty Queue (Q))
{
  v = DeQue (Q);
  pcolor [v] = white;
  for u ∈ Adj [v]
  if pcolor [u] = Black;
  {  pcolor [u] = Grey;
     Enqueue (Q, u);
  }
}
```

white = completely visited
grey = in the queue
Black = not visited

Date: __/__/____

Code :- (My Implementation)

```
Q = CreateQueue (n);
Enq (Q, s);
while (!EmptyQue (Q))
{   int x;
    x = Deque(Q);
    pcolor[x] = white;
}
graph temp = G -> vertex[x];
while (temp -> next != NULL)
{
    if (pcolor[temp -> next -> vertexID] == black)
    {   pcolour[temp -> next -> vertexID] = Grey;
        Enque (Q, temp -> next -> vertexID);
    }
    temp = temp -> next;
}
}
```

Date: __/__/__

```
void BFS ( Graph G)
{
  int i, N;
  int * pcolor;
  Vertex u, v;
  Que Q;
  Node temp;
  N = G → iN;
  pcolor = (int *) malloc ( N * sizeof (int));
  for ( int i = 0 ; i < N ; i++)
  {
     u = G → pvertex[i] → VertexID
     pcolours[u] = BLACK;
  }
  pcolour[s] = GRAY ; // gray nodes are added to the queue
  Q = CreateQue(N);
  Enque (Q, (QElement) s);
  while ( ! IsQueEmpty (Q))
  {
     u = Deque (Q);
     temp = G → Vertex[u];
     while ( temp → pnext != NULL)
     {
        v = temp → pNext → VertexID;
        if ( pcolour[v] == BLACK)
        { pcolour[v] = GRAY;
        } Enque (Q, (QElement) v );
```

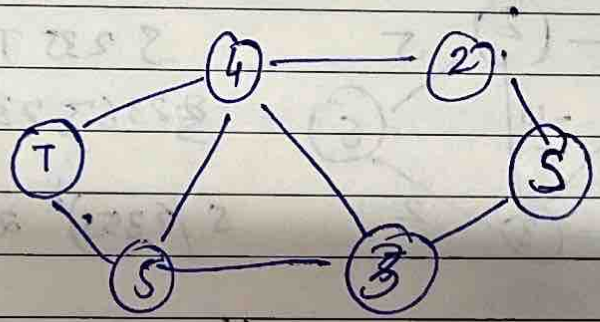
Date: __/__/____

```
        temp = temp → pnext ;
    }
    pcolour[u] = WHITE ;
}
return ;
}
```

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

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

Shortest Path Problem :-



Possible paths:-
S 2 4 T
S 2 4 5 T
S 3 4 T
S 3 5 4 T
S 3 4 5 T
S 2 4 3 5 T

- Path is shortest from S to T if the cost to travel is minimum, if no path then cost is ∞.
- Given a graph G(V, E) with a weight/cost function w

Date: __/__/__

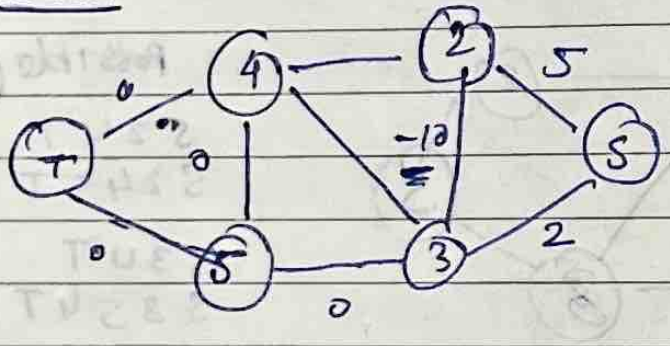
- cost of path $p = (v_0, v_1, \ldots v_k)$
  $c(p) = c(v_0, v_1) + c(v_1, v_2) + \cdots + c(v_{k-1}, v_k)$

- SPP
  - given $s, t \in V$
  - $\delta(s,t) = \min \{ c(p) \mid p \text{ is a path from } s \text{ to } T \}$
    $= \infty$ if no path

- A shortest path from s to t is any path p s.t. $c(p) = \delta(s,t)$

- we define shortest paths only if there are no negative edge cycles.

Reason



S 235 T -5

S 2355235 T -8

S $(2355)^k$ 235T

thus, no such k can be determined.

Exploring via destination

* shortest path from s, T;

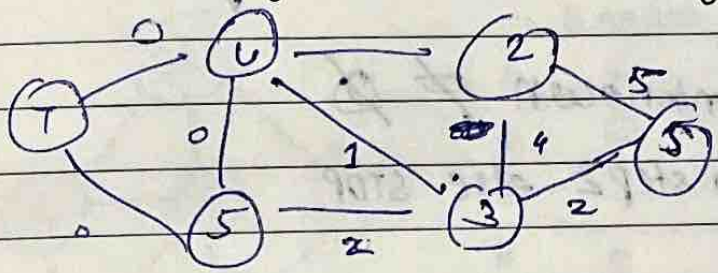
$\delta(s,T) = \min \begin{cases} \delta(s,4)_{min} + c_{4T} \\ \delta(s,5)_{min} + c_{5T} \end{cases}$

Date: __/__/____

The above may look easy, but challenge occurs when the shortest path to t_4 include $5 \to \infty$ (or vice versa), thus; the combinations you will have to explore would be large.,

Thus, we will explore from the source.

So u are updating the costs as you proceed.



DIJSKTRA algorithm LOGIC -

- No (-ve) edge weights / costs

d_u: cost of shortest path (current) from ∞ to u from s.

We are gonna find the shortest path from 1 to all other nodes.

Step-1

$d_1 = 0$, $d_j = \infty$ $\forall j \neq 1$

known [] = 0

$K = \emptyset$

→

Date: __/__/__

Step-2

i ← argmin
j ∈ unknown d_j
known[i] = 1 (k = i ∀ i)

Step-3

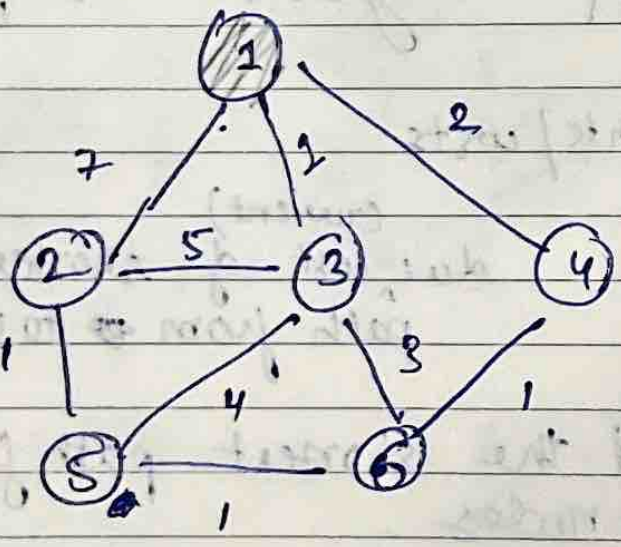
$\forall j \in Adj[i]$ & j = unknown

pv = i

$d_j = \min\{ d_j, d_i + c_{ij} \}$

Step 4

if unknown ≠ ∅
go to step2 else STOP



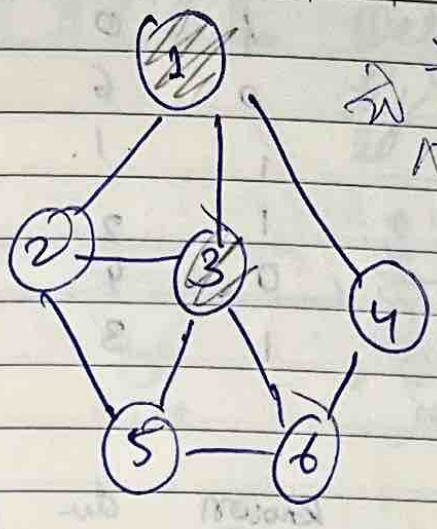
	known	d_v	pv
1	1	0	0
2	0	∞	0
3	0	∞	0
4	0	∞	0
5	0	∞	0
6	0	∞	0

ultimate shortest path
पता है कि we will
mark known as 1

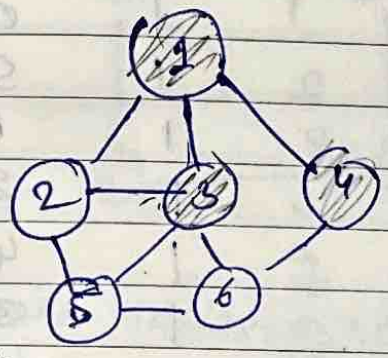
pv ⇒ current shortest

we have to select, from that which has shortest degree amongst the unknown

Date:



	known	dv	pv
1	1	0	0
2	0	7	1
3	1	1	1
4	0	2	1
5	0	∞	0
6	0	∞	0

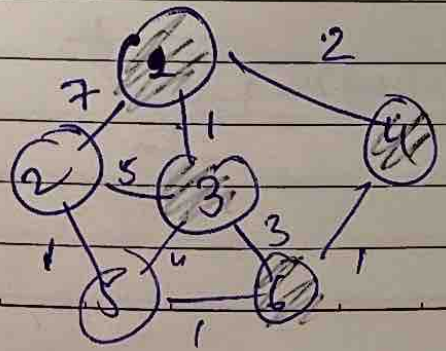


	known	dv	pv
1	1	0	0
2	0	6	3
3	1	1	1
4	0	2	1
5	0	5	3
6	0	4	3

for eg:

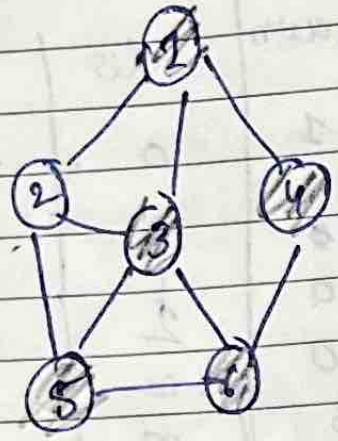
~~u can't reach~~ in unknown set, whichever has least degree, उससे better you can't do.

pv = previous vertex ⇒ कौनसे vertex से तुमने ये ~~shortest~~ current dv / current shortest path calculate किया है

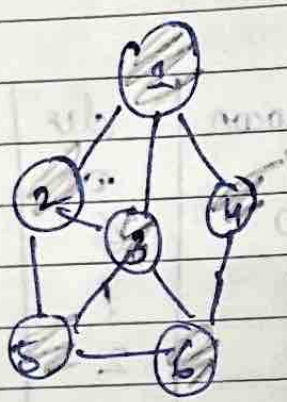


	known	dv	pv
1	1	0	0
2	0	6	3
3	1	1	1
4	1	2	3
5	0	5	4
6	0	3	

Date: ___/___/___



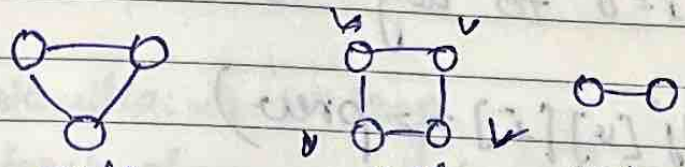
	known	dv	pv
1	1	0	0
2	0	6	3
3	1	1	1
4	1	2	1
5	0	4	6
6	1	3	4

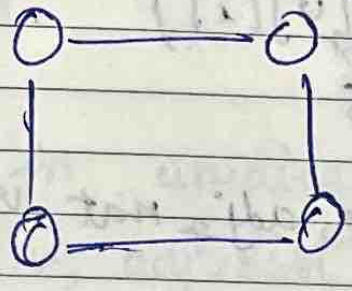


	known	dv	pv
1	1	0	0
2	1	5	5
3	1	1	1
4	1	2	1
5	1	4	6
6	1	3	4

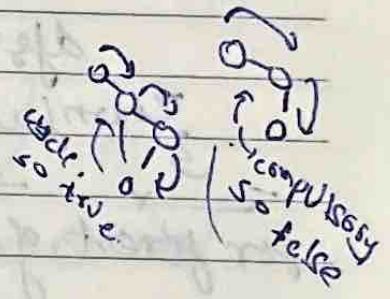
Date: __/__/____

CYCLE Detection IN UNDIRECTED graphs

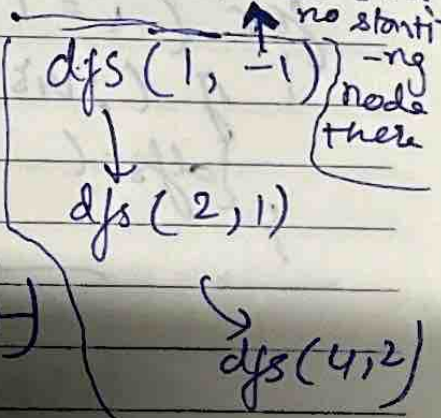




LOUV (youtube) पर



You are visiting a node;
↳ not prev node
↳ already visited.



~~adj. matrix~~

~~vis [n] = false~~

~~dfs (v, prev, adj_mat, vis)~~

~~dfs (~~

~~{~~

~~if (vis[v])~~ // cycle can only be detected if you are visiting an already visited node.

~~}~~

1 → 2, 3 4 → 2, 3
2 → 1, 4
3 → 1, 4

Date: __/__/__

```
bool dfs ( v, prev, adj_list, vis )
{
    visit[v] = true ;
    for ( int i=0 to adj_list[i] size ())
    {
        if (adj[v][i] == prev)
            continue;

        if ( vis [ adj[v][i] )
            return true;

        dfs (adj[v][i], v, adj, mat, vis) ;
    return false;
}
```

for forest of graphs of unconnected graphs.

```
for (i = 1. to n)
{ if (!vis[n])
    {dfs( )}run कर दो
}
```

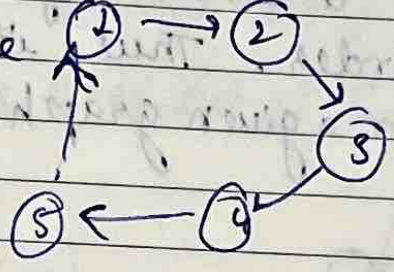
```
if (bodret == true)
    return true;
```

Date: __/__/____

vis = {false}
path_vis = {false}

in a directed graph to detect cycles we will use two visited arrays.

Basic idea is to keep a track of every node that you have visited.

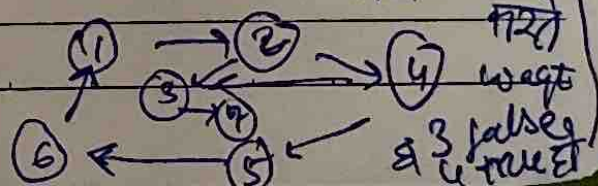


// cycle in unidirected graphs :-

```
bool dfs ( v, adj-list, vis, path-vis )
{
    if (path_vis [v]) ⇒ cycle has been detected
        return true ;

    if (vis [v])
        return false ;

    vis [v] = True ;
    path-vis [v] = True ;
    for ( i = 1 to adj-list [v].size ())
    {
    bool ret = dfs (adj-list [v] [i] ; adj-list, vis, path-vis);
    }
    path [v] = false ;
}
```

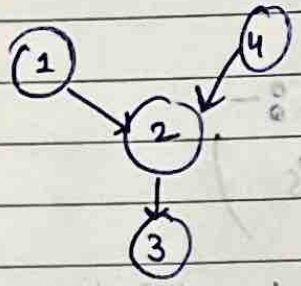


Date: __/__/__

TOPOLOGICAL sorting

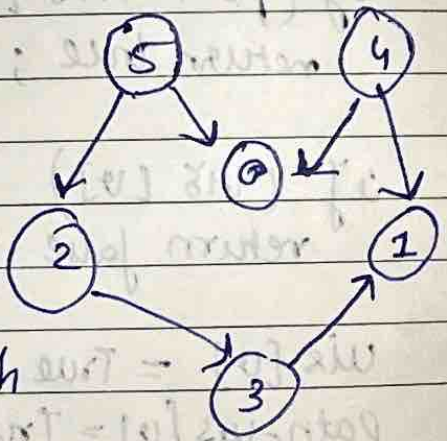
↳ u can only do it on a directed graph that is acyclic.

$u \to v \Rightarrow$ u comes before v in the sorted/final order. Thus, it may not be unique for a given graph.



Ans. 1423 } both are
4123 } acceptable

Rule of Thumb :- Always start with vertex with indegree = 0.



Methods :- (a) Take 2 visited arrays
(b) Go with stack based approach

(u) → (v) → (w) → (z)

u, v, w ≥ 1

thus we are travelling decern, Thees, somewhat modification of DFS

```
dfs ( adj-list, vertex v, vis, stack)
{
    vis[v] = true;
    for ( int child = v )
    {
        if ( vis[child] == false)
        {
            dfs (adj-list, child, vis, stack)
        }
    }
    stack.push (v)
}
for (i=0 ; i<n , i++)
{
    if (vis[i] == false)
        then dfs
}
```

denotes the present

Ans: 5 4 2 3 1 0

dfs(u)
dfs(v) dfs(z)

pop the stack till empty

5
4
2
3
1
0

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

dfs(0)
↓ no kids ⇒ push
ends

dfs(1)
↓ no kids
ends

dfs(2) ← no more child of 2 ⇒ 2 pushed
↓
dfs(3) ← 3 pushed
↓
dfs(2)

① If children nahi ⇒ push it

1 If children nahi ⇒ push it

Date: __/__/__

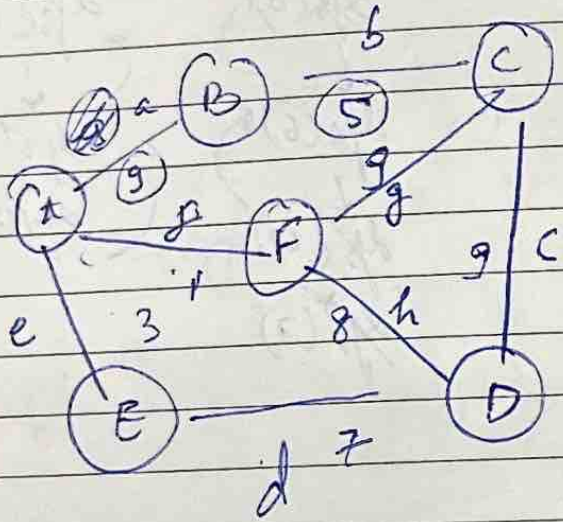
MST

(a) All nodes should be connected
(b) connected
(c) doesn't contain a cycle

} Tree

↓

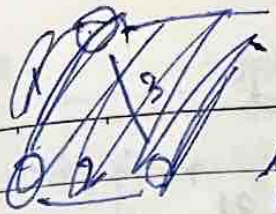
it shd satisfy these properties



(a) sort (use maxheap) → edges
(b) delete edges in decreasing order as long as it remains a completed graph

a bunch of disconnected tree is a forest.
a single tree is also a forest

(a) sort
(b) keep adding in incr. order as long as ensuring cycles do not form.



Date: ___/___/____

if adding an edge connects two already connected edges then we don't add it, we only connect disconnected trees/nodes.

↳ network optimisation # MST

edges designing edges.

→ → → →

DIJKSTRA's shortest Path (S, *) Algorithm.

dv ⇒ stores the currently known shortest path for each node

Analysis:- Step1 ⇒ O(N) initialisation

Step-3 ⇒ O(E) work done :: u visit all edges to update & make it known

Step-2 ⇒ To find min, you visit each node each time.

⇒ V+(V-1)+(V-2)+.... ⇒ $|V|^2$ work.

In Heap, to decrease the key/priority, store the position also. O(logn)

⇒ ($|V|+|E| \log |V|$)

log

Date: __/__/__

if min^m is extracted.
all nodes are queued, we update it's neighbours if there is shorter dist - if update.

→ Total work done if you are using fibonacci heaps, will be $|V| \log |V| + |E|$

Priority Queue
- Extract Min
- Decrease Key

```
Initialise(G,s);
S:= ∅;
Q:= V[G];
while Q ≠ ∅ do
u: Extrad-Min(Q);
S = S ∪ {u}
for each v ∈ Adj[u] do
    Relax(u,v,w)
od
od
```

```
dv = min(dv, du) + w uv
( dv → du + w uv
is if the newer dv is shorter then we updated.
```

pds ⇒ Do distance_to_source.

Code

Date: __/__/____

```
# define INFTY 1000
# define UNKNOWN -1
# define NOTVERTEX -1

struct stSPTable
{
    int * pKnown;
    float * pds;
    Vertex * pp;
};
```

pp = pPrevious_vertex.
pds = on previous page written

Date: __/__/__

```
SPTable Dijkstra (Graph G, float **c)
{
  SPTable myTable;
  int i,j,N;
  PQue Q;
  Vertex u,v;
  Node temp;

  //Allocate memory,
  N = G->iN;
  myTable = (SPTable *) malloc (sizeof (struct SpTable));
  if (myTable == NULL)
  {
    exit(0);
  }

  myTable->pKnown = (int *) malloc (sizeof (int));
  myTable->pds = (float *) malloc (N * sizeof (float));
  myTable pp = (Vertex *) malloc (N * sizeof (Vertex));

  // Initialise table

  for (int i=0; i<n; i++)
  {
    ~~String~~ ...   ~~myTable->pKnown[i] = 0;~~
    myTable->pKnown[i] = 0;
    myTable->pds[i] = c[i][0];
    myTable->pp = UNKNOWN;
  }
```

Date: ___/___/____

// create PQue.

Q = Create PQue(N);
for (i=0; i<N; i++)
    PEnq (Q, (Element) i, (Key) myTable → pds[i]);

// while there is node for which shorts path not known.
while (! IsPQEmpty (Q))
{
    u = (vertex) Extract min (Q);
    myTable → pknown[u] = 1;
    temp = Q → pVertex[u]

    while (temp → pNext != NULL)
    {
        v = temp → pnext → vertex;

        if ((! myTable → pknown[v]) &&
            (myTable → pds[u] + c[u][v]
                < myTable → pds[v])
        {
            // Relax opn.
            myTable → pds[v] = myTable → pds[u] + c[u][v]
            myTable → pp[v] = u;
            DecreaseKey (Q, (Element) v, (Key) myTable → pds[v]);
        }
        temp = temp → pNext
    }
}

in practicals we will use this. We can't run from scratch

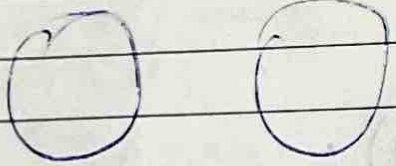
Date: __/__/__

BELLMAN FORD (out of syllabus. so learn it fck it)

→ advantages:-
⇒ -ve edge
⇒ supports dynamic updates

MINIMUM Spanning Tree :-

① our goal is to make the shortest path same.



→ KRUSKAL'S Algo.

② Theorem :- (Cycle Property of MST)

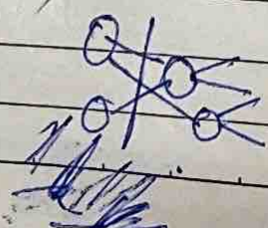
For any cycle C in a graph the heaviest edge doesn't appear in MST.

→ Prim's Algo.

③ Theorem :- (cut Property)

For any proper non-empty subset X of vertices the highest edge with exactly one point in X belongs to MST

Tree → collection of connected



cut is collection of edges s.t. if u remove those edges, you get two disconnected components.