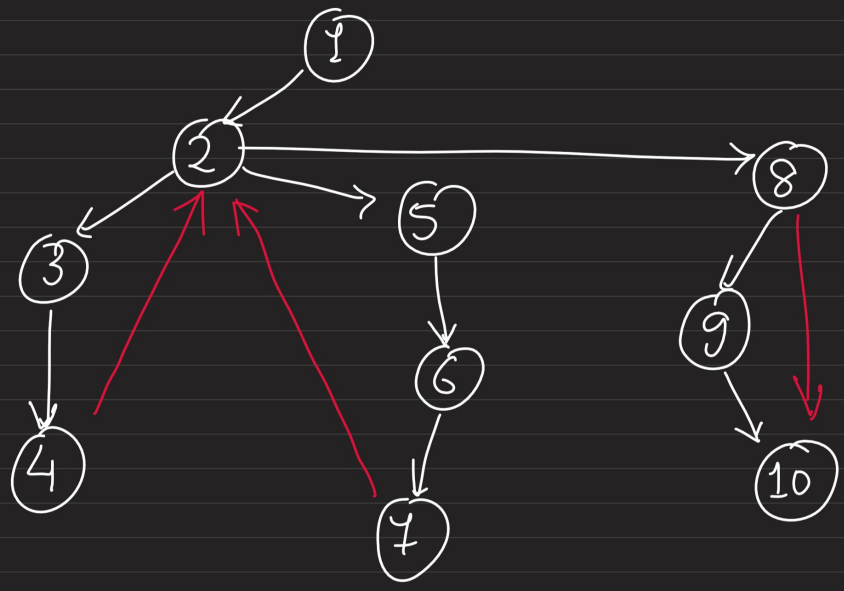
Graphs :

Cycle detection

↳ printing cycle using DFS.



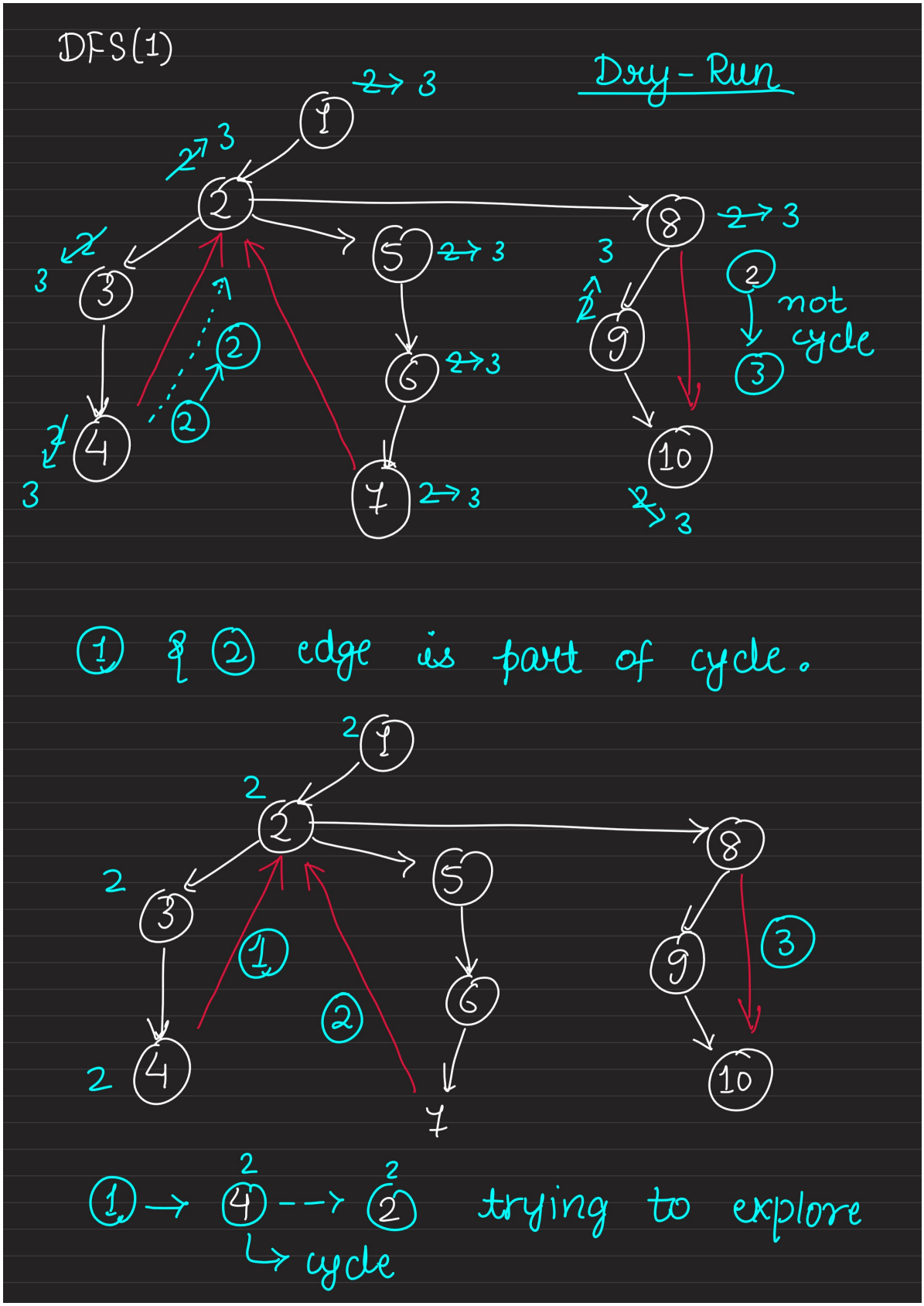
—— not visited in DFS

—— visited during DFS

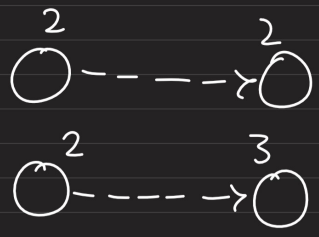
```
col[all] = 1;
DFS(v)
{
    col[v] = 2
    for (x ∈ neig[v])
        if (col[x] == 1)
            DFS(x);
    col[v] = 3;
}
```

We will call DFS(1)

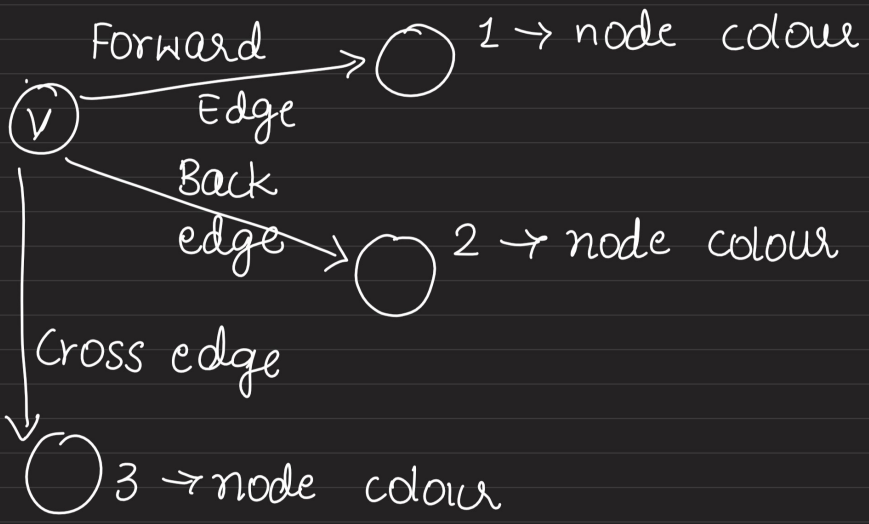
- 1 → not visited
- 2 → visited but neighbours not visited
- 3 → visited all neig.


DFS(1)
Dry - Run
not cycle
① & ② edge is part of cycle.
① → ④ --> ② trying to explore
↳ cycle

Unex. Edge b/w 2-2 colour forms a cycle
" " b/w 2-3 colour doesn't.



Concept:



Back Edges creates cycle.

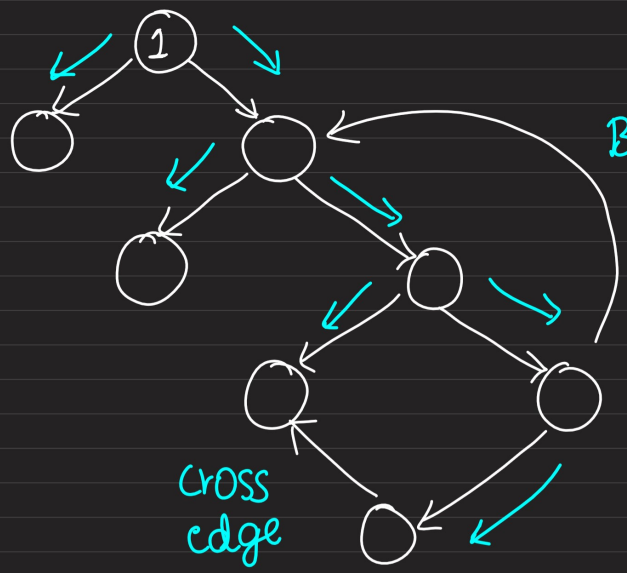
```cpp
void dfs(int node){
    col[node]=2;
    for(auto v:g[node])
    {
        if(col[v]==1) //node -- v is forward edge
        {
            dfs(v);
        }
        else if(col[v]==2) //node -- v is back edge
        {
            //we found a cycle
            is_cycle=1;
        }
        else if(col[v]==3) //node -- v is cross edge
        {
            //not of any importance
        }
    }
    col[node]=3;
}
```

code for check cycle.

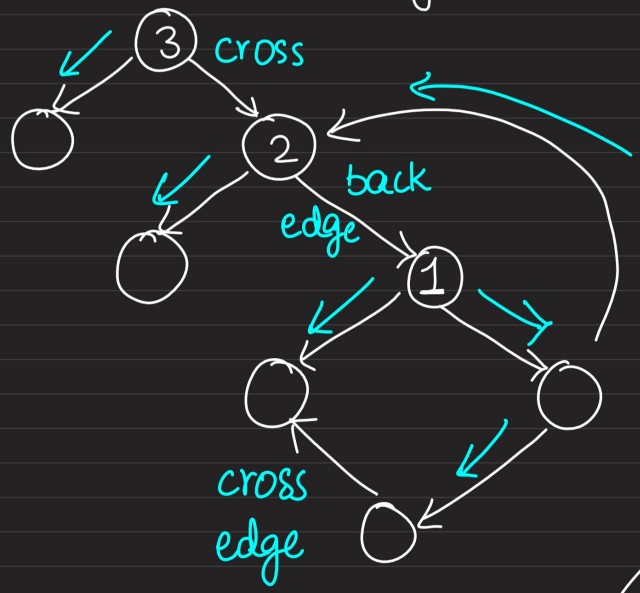
```cpp
void solve()
{
    //Inputting Directed Graph
    lli n,m;cin>>n>>m;
    g.resize(n+1);
    col.assign(n+1,1);

    for(int i=0;i<m;i++){
        int x,y;
        cin>>x>>y;
        g[x].push_back(y);
    }

    //visiting all nodes iterate through all nodes
    for(int i=1;i<=n;i++)
    {
        if(col[i]==1) //dfs on unvisited nodes
        {
            dfs(i);
        }
    }
}
```

as graph is not connected.

- dfs(1) explores all nodes



Different starting :



dfs(1)
dfs(3)

needed to call to explore all edges

that's why dfs on all nodes are important.

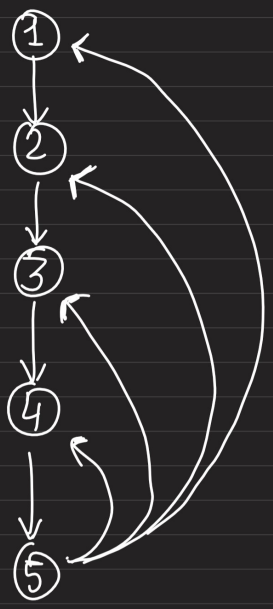
How to print cycle :-

We will create a parent of all node.

```cpp
vector<vector<int>> g;
vector<int> col;
vector<int> parent;

bool is_cycle = 0;
vector<int> any_cycle;

void dfs(int node,int par){

    parent[node] = par;
    col[node]=2;
    for(auto v:g[node])
    {
        if(col[v]==1) //node -- v is forward edge
        {
            dfs(v,node);
        }
        else if(col[v]==2) //node -- v is back edge
        {
            //we found a cycle !!
            //storing the first cycle
            if(is_cycle==0) //we will find the chain of parents
            {
                //storing nodes that create a cycle
                int temp = node;
                while(temp!=v)
                {
                    any_cycle.push_back(temp);
                    temp = parent[temp];
                }
                any_cycle.push_back(temp);
            }
            is_cycle=1;
        }
        else if(col[v]==3) //node -- v is cross edge
        {
            //not of any importance
        }
    }
    col[node]=3;
}
```

Time complexity : $O(V+E)$

We cannot use this algo to print all the cycle.

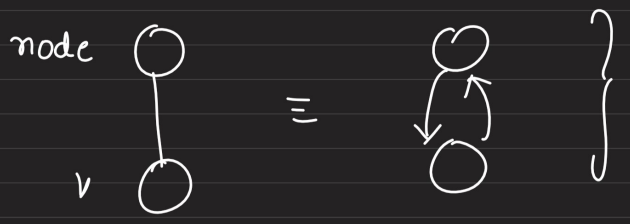


$O(N^2)$

#
no. of nodes to be stored = $O(N^2)$

We can use same algorithm for undirected graph as well.

1 problem



this will get counted as a loop.

It will detect everyedge as loop.

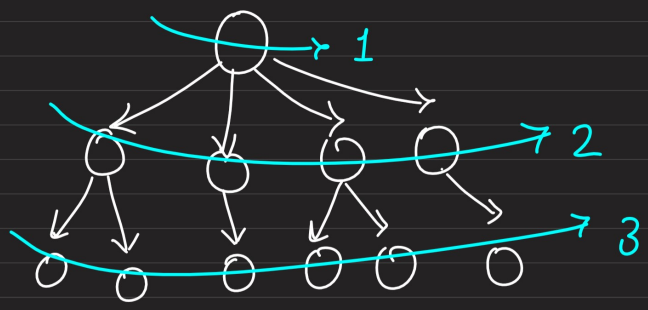
```
if (par[v] == node) continue;
```

↳ don't consider that as loop. or neighbour

{ If there are self loop or multiple edge the handle them separately.

BFS :-

Breadth first Search



level by level explorati.

We explore nodes which are closest to the parent node

can be used for shortest path probl.

more powerful

BFS >> DFS → recursion (stack)

easy to code.

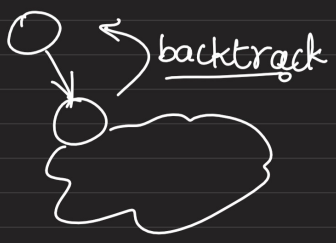
automatically theire in recursion

we will use queue

we will have to implement

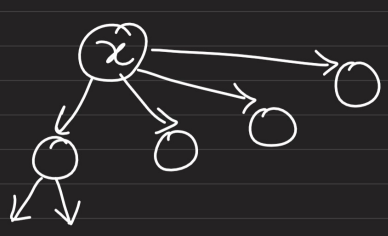
#Revelwing DFS :

```
stack ds;
x -> ds
for (v: g[x])
    explore(v);
```



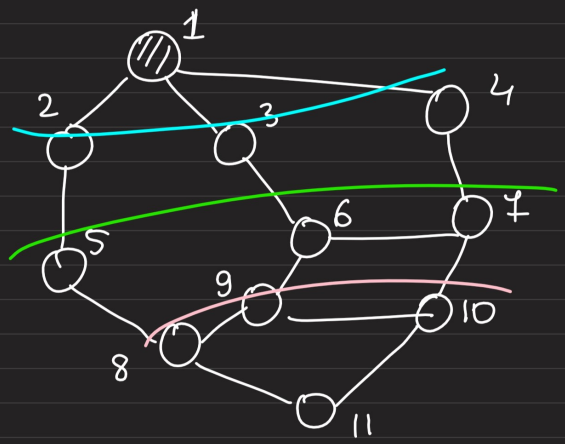
BFS

```
queue ds;
```




```
remove x from ds;
while (ds !empty)
{
    x = ds.front();
    ds.pop()
    for (v : g[x])
        ds.add(v)
}
```

Dry - Run

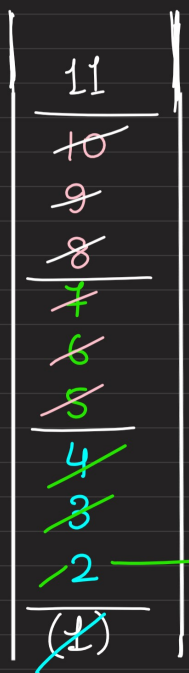


—— : distance 1 from node (1)

—— : distance 2 from node (1)

—— : distance 3 from node (1)

visited[]++ →

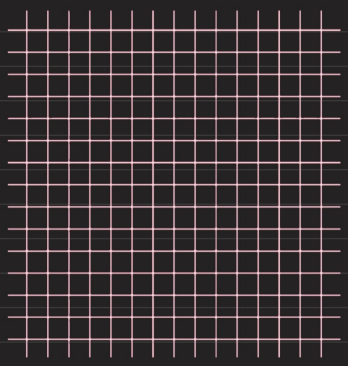


2 will get deleted while adding 5 to que

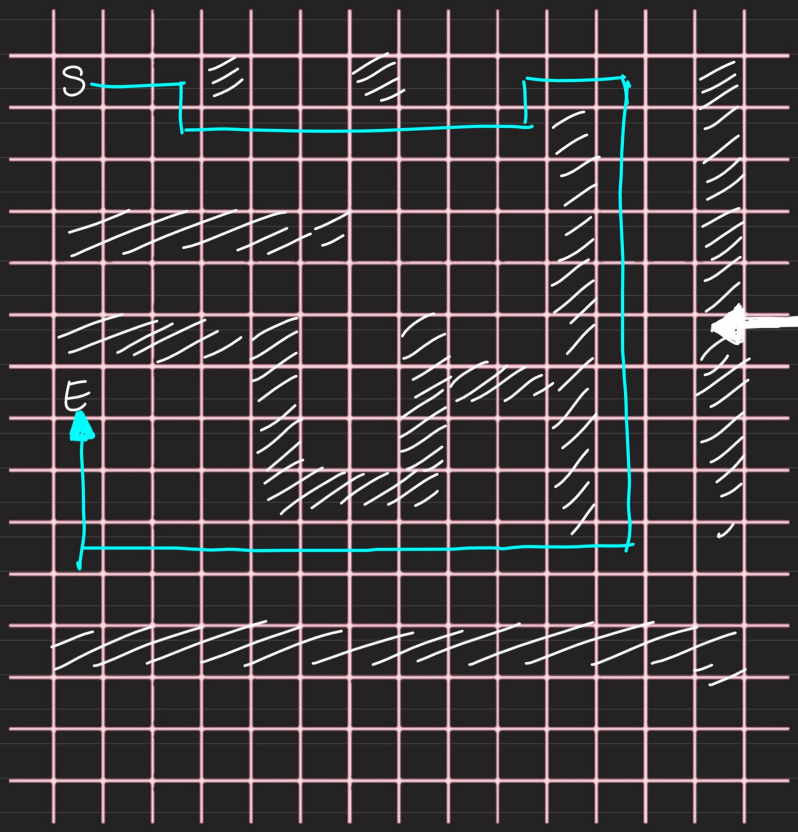
NOTE: for unweighted graph, distance b/w 2 nodes is the minimum no. of edges b/w them.

Graph

- Explicit
  - eg: General graphs.
- Implicit
  - eg: Grid.



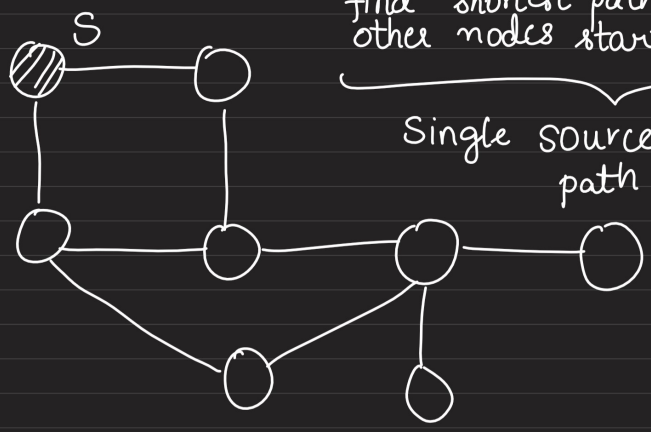
1.



Find the minimum no. of moves to reach E from S.

NOTE: DP on grid can be used for DAG. when → only 2 → types of moves are allowed.

Shortest Path :



find shortest path to every other nodes starting at s.

Single source shortest path

SSSP

Weight of all edges = 1/unweighted
↳ BFS (Queue) } O(V+E)

W = 0/1 ↳ 0/1 BFS (Deque) }

W = [0,∞) → Dijkstra Algorithm (Priority Q) } (V+E) log V

W = (−∞, ∞) → Bell-man ford (DP) } O(V·E)

APSP (All pair shortest path)
↳ Floyd Warshall. (Matrix) } $O(V^3)$

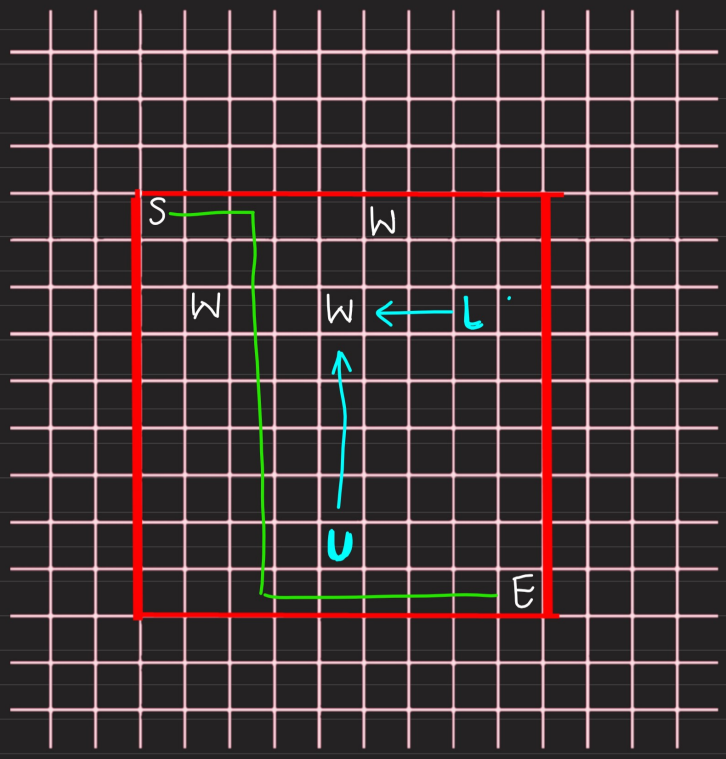
According to weights, you will be using different algorithm.

Steps

i) Recognize and Reduce the problem to shortest path problem

ii) Decide Algo based on weights.

Q. N, M matrix



↳ line of view of police

W - wall

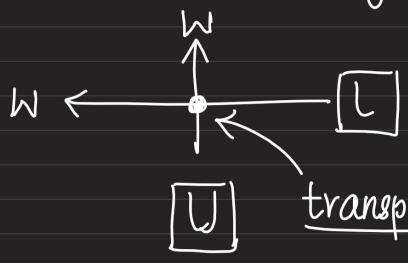
U/D/L/R

Police view direction

You need to go to E without coming into vision of police.

soln:

- Make the cells that police sees as walls but transparent so that other police can see through it.



Time complexity ?? $O(N \cdot M \cdot \max(N,M))$

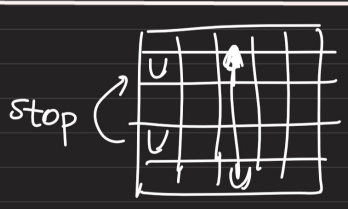
We cannot mark each of the cells in the vision of police.

Optimization :

stop processing hear only.

Imp.

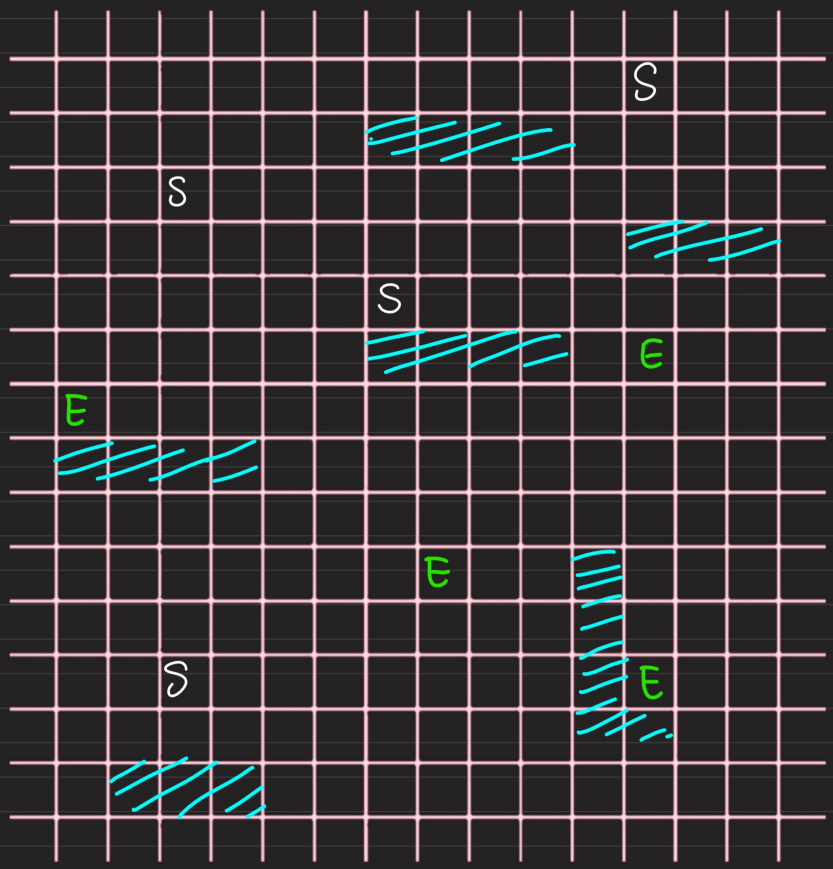
we can armotize our code to N × M.



for each vision we will traverse atmost N·M cells.

MSSP

Q1



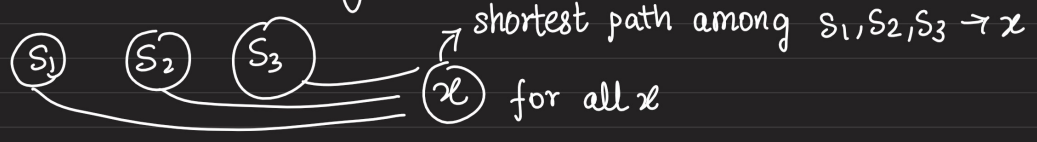
For every start, find me the closest exist.

soln: Naive : TC : $O(\#S \cdot N \cdot M)$

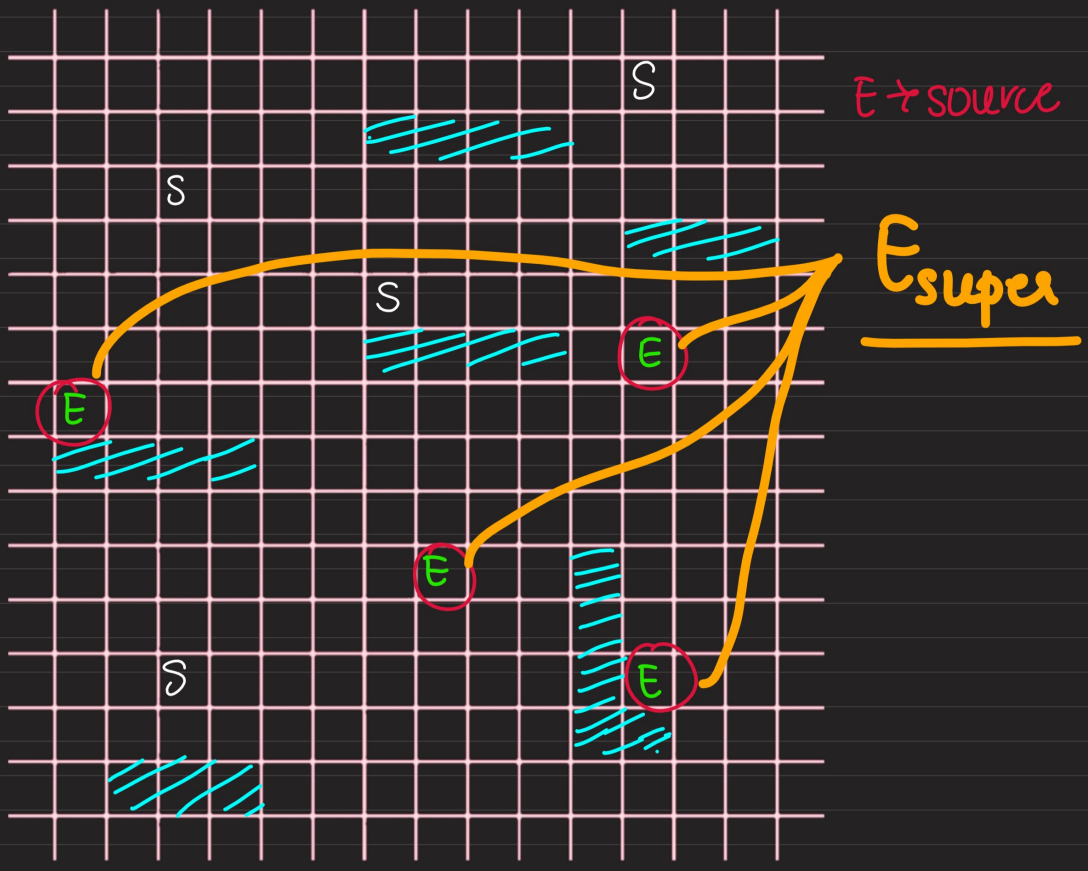
How to do it in $O(N \cdot M)$

$$\left\{ ans[i] = \min_{1 \le j \le |E|} (S_i, E_j) \right\}$$

We will be using MSSP.



We will keep exit as source.



for any S, find the closest source node.

How to solve MSSP problem?

i) Create End Super

ii) Run BFS on end-super.

* If we go from E_{super} to (S), the route will be $E_{super} \rightarrow E_{\text{closest to S}} \rightarrow S$

$dist(E_{sup}, S) - 1$ = distance b/w S & its closest E

E_{super} → first push all E_j into priority queue and update their distance as 1.

So, instead of creating E_{super}, we will manually do it. } **

* $Ans_i = \min_{j \le |E|} (S_i, E_j)$

↳ put all E_j in the Queue of BFS with distance 0.

↳ the run BFS

↳ distance array you get contains, distance of that node from the closest E_i.

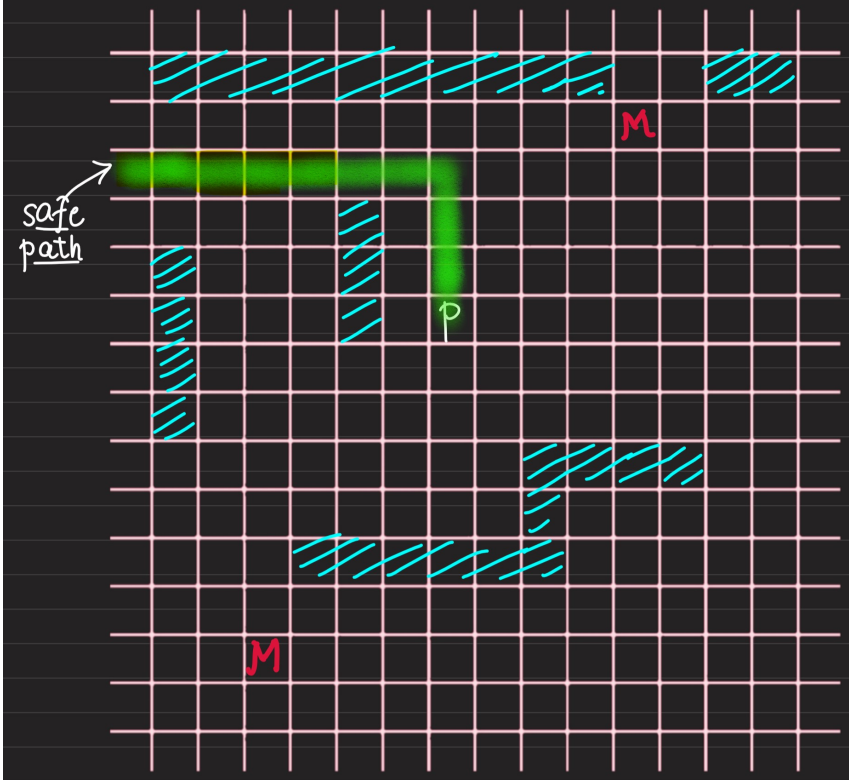
Multisource BFS idea

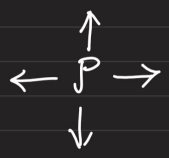
Since, we are doing BFS only once, so

TC = $O(N+M)$

Multisource Shortest Path :-

Q2 Save Yourself.





1) Take grid
2) Fix path for P
3) Monster will chase P

} Restriction

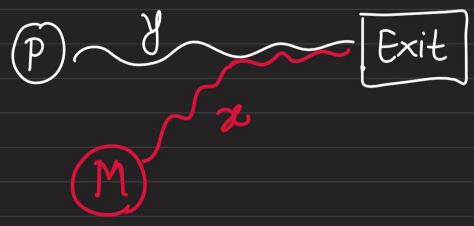
Output : Can you move out of the grid?

- Monster can wait if they intercept the path before you.

for $x \in p_{optimal}$, $dist(P, x) < dist(M, x)$

↳ But we don't know the optimal path.

Observation :



If monster can reach exit b4 person, the person will not be able to exit.

$$dist(P, E_i) < \min_{j \le |M|}(dist(M_j, E_i))$$

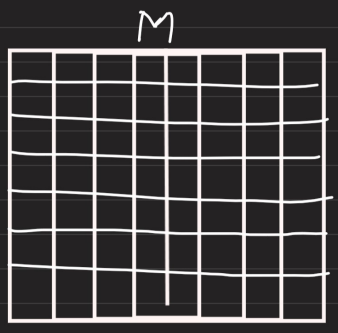
relaxed condition.

$O(N \cdot M)$ $\quad$ $O(x \cdot NM)$

for each M we will find dist.

P → person

x → # monster



Total time complexity

$O(x \cdot N \cdot M)$

```
for exit[i]
    for (j in range [1, x])
        min(dist(Mj, Ei))
if (dist(P, Ei) < min(dist(Mj, Ei)) "Yes"
```

T.C $O(N+M) \cdot x$

Total TC : $O(x \cdot N \cdot M + (N+M)x)$

$\equiv O(x \cdot N \cdot M)$

Worst case : $O(N^2 M^2)$ as $0 \le x \le NM$

How can we do better ?

#exits < #monsters

for (exit[i])

BFS(E_i) $\rightarrow O(N \cdot M)$

- dist(P, E_i)
- min(dist(M_j, E_i))

dist(P, E_i) = dist(E_i, P) — Bidirectional Relation

Time complexity : $O((N+M)\,NM)$.

Can we do better than this ?? YES

We only want the monster closest to a particular exit.

- possibility of doing better.

↳ we can use, multisource BFS with all the Monsters. M_{super}

give me how much dist. does M closest to Ei have to travel.

So, put all the monsters in the queue with distance '0'.

Dry Run :

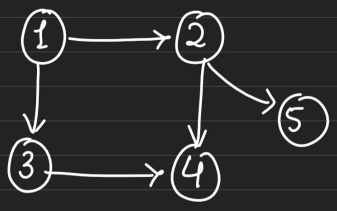
n = 5 , m = 8

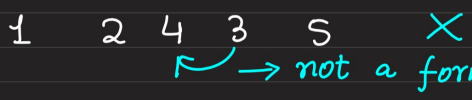
→ wall
A → start
. → floor
M → monstes

Topological Ordering

Ordering nodes of a directed Acyclic graph.

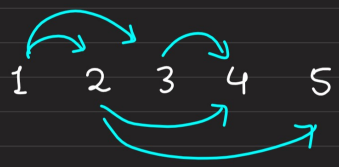
Condition
→ Edges go forward in direction.



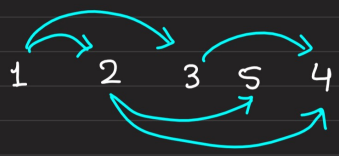


→ not a forward edge

Is Topological Ordering unique ? Ans → No



} we can alternate 2 & 3 as well



Using DFS O(V+E)

```
vector<vector<int>> g;
vector<int> vis;
vector<int> topo;

void dfs(int node)
{
    vis[node]=1;
    for(auto v:g[node])
    {
        if(!vis[v])
        {
            dfs(v);
        }
    }
    topo.push_back(node);
}
```

every neigh. of node will get pushed in this part.

then node will be pushed.

```
void solve()
{
    lli n,m;cin>>n>>m;
    g.resize(n+1);
    vis.assign(n+1,0);

    for(int i=0;i<m;i++)
    {
        int a,b; cin>>a>>b;
        g[a].push_back(b);
    }

    for(int i=1;i<=n;i++)
    {
        if(!vis[i]){
            dfs(i);
        }
    }
    reverse(topo.begin(),topo.end());
    for(auto v: topo)
    {
        cout<<v<<' ';
    }cout<<'\n';
}
```

reverse to get the correct ordering

Longest Path in a DAG

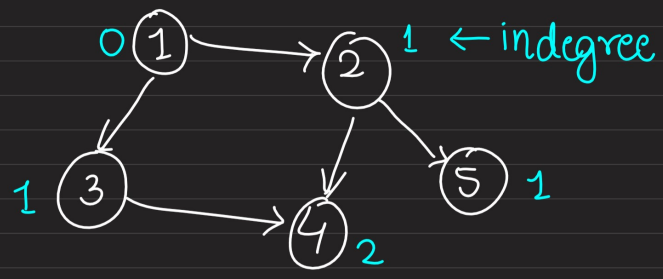
Using DP

```
//DP can be used only if their is no cycles
int dp[100100];
int rec(int node) //longest path starting at node
{
    if(dp[node]!=-1) return dp[node];
    int ans =1 ;
    for(auto v:g[node]){
        ans = max(ans,1+rec(v));
    }
    return dp[node] = ans;
}
```

```
void solve()
{

    lli n,m;cin>>n>>m;
    g.resize(n+1);

    for(int i=0;i<m;i++)
    {
        int a,b; cin>>a>>b;
        g[a].push_back(b);
    }
    memset(dp,-1,sizeof(dp));
    int ans = 0;
    for(int i=1;i<=n;i++)
    {
        ans = max(ans,rec(i));
    }
    cout<<ans<<'\n'; //returns no. of nodes in the
}
```

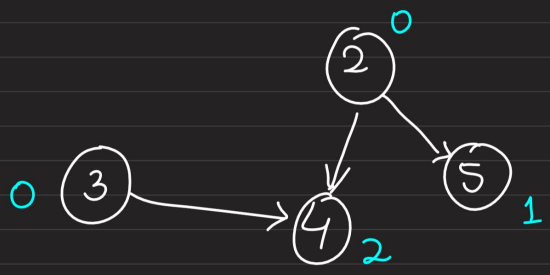
Using BFS (>> DFS)

(Kahn's algorithm)



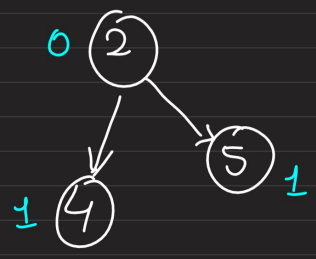
Algo: Find the node with indegree = 1, then delete it. Keep doing it.

Step 1: [1]



Step 2: [1 3] for multiple possibilites take any one

3 taken



Step 3: [1 3 2]

Step 4: [1 3 2 4 5]

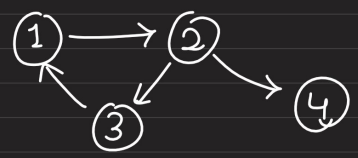
```cpp
void kahn()
{
    queue<int> q;
    for(int i=1;i<=n;i++){
        if(indeg[i]==0)
        {
            q.push(i);
        }
    }
    while(!q.empty()){
        int cur = q.front();
        q.pop();
        topo.push_back(cur);

        //instead of deleting the
        for(auto v : g[cur])
        {
            indeg[v]--;
            if(indeg[v]==0){
                q.push(v);
            }
        }
    }
}
```

instead of deleting the node, we update indeg array

NOTE: If the graph contains cycle, then after some operation, we will be left will nodes where none will have indeg = 0.

so, topo.size() < n } if graph contains cycle

get deleted, after that Kahn() will terminate.

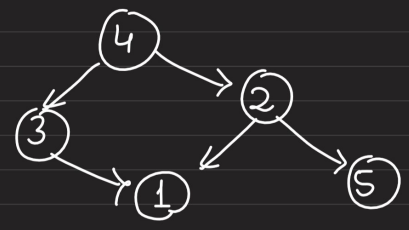


no node with indegree = 0

Use BFS mainly.

Q Find lexiographically smallest topological ordering.

soln:



[4 2 3 1 5]

3,5 1,5

2, 3

Whenever there is possibility of getting smaller no. take it.

We always need to get the smallest one among all with '0' degree.

{minimum priority queue.}