DFS → Comp no
DFS → Alternate Coloring / Bipartite
DFS → cycle

Flood Fill
graph is array

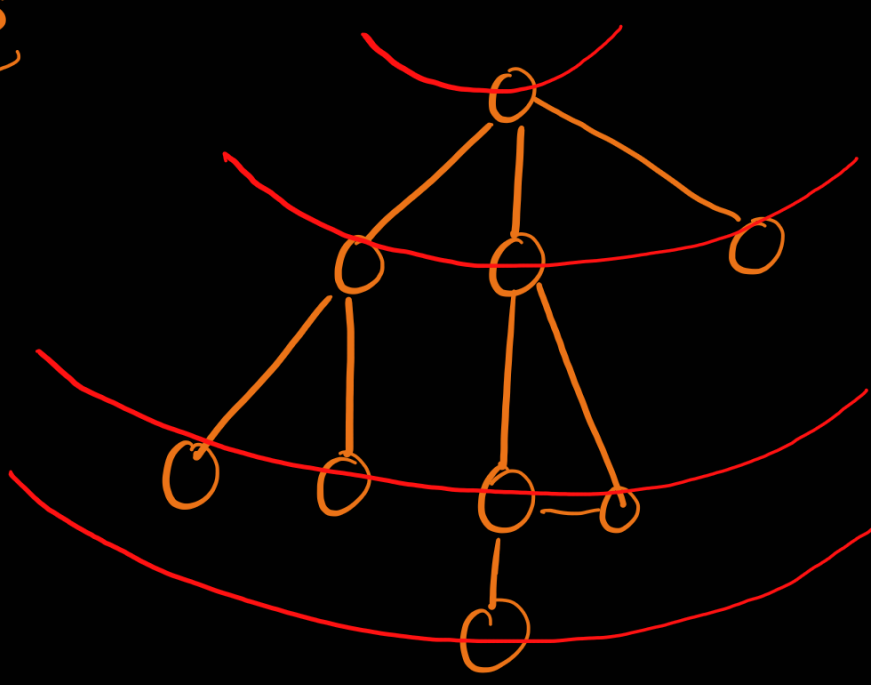
Shortest Path Problems

BFS → Topological Sort
BFS → SP on graphs where all edge are same

Modelling
- 0-1 BFS
- 0-∞ Dijkstra
- -∞-∞ Bellman Ford
- Floyd Warshall → All pair shortest path

SSSP - Single source shortest path

BFS

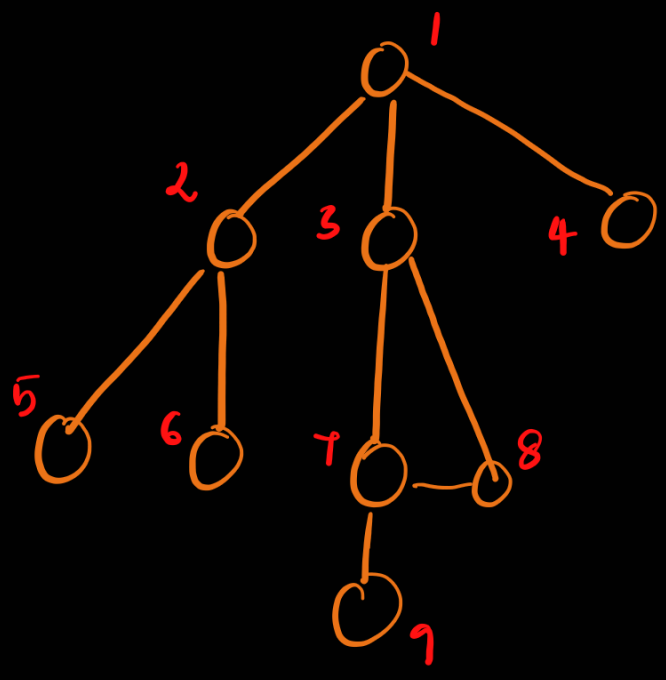


Explore layer by layer

Uses queue

DFS :- Recursion, Stack

AlgoZenith



Queue things to be done later on

1 visited

1 [2 3 4]

2 visited

2 3 4 [5 6]

3 visited

3 4 5 6 [7 8]

4 visited

4 5 6 7 8

5 visited

5 6 7 8

6 visited

6 7 8

7 visited

7 8 [8 9]

AlgoZenith

8 visited

8 8 9

8 already visited, skip

8 9

9 visited

9

Done

CSES Labyrinth

You are given a map of a labyrinth, and your task is to find a path from start to end. You can walk left, right, up and down.

Input

The first input line has two integers n and m: the height and width of the map.

Then there are n lines of m characters describing the labyrinth. Each character is `.` (floor), `#` (wall), `A` (start), or `B` (end). There is exactly one `A` and one `B` in the input.

Output

First print "YES", if there is a path, and "NO" otherwise.

If there is a path, print the length of the shortest such path and its description as a string consisting of characters `L` (left), `R` (right), `U` (up), and `D` (down). You can print any valid solution.

Constraints

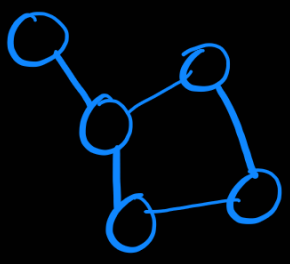
- $1 \le n, m \le 1000$

Example

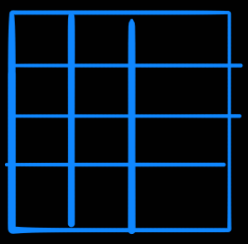
Input:
```
5 8
########
#.A#...#
#.##.#B#
#......#
########
```

Output:
```
YES
9
LDDRRRRRU
```

AlgoZenith



Explicit Graph



Implicit Graph

Rules defined on edges

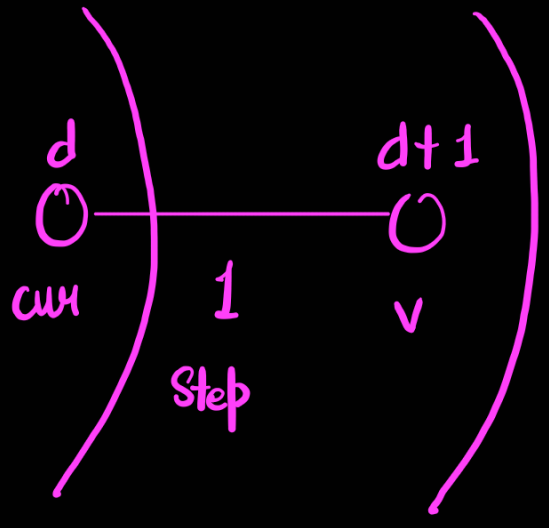
Node - cell (i, j) uniquely defines my position

State - What defines our node

Find Start, End State

Initially everything at INF dist

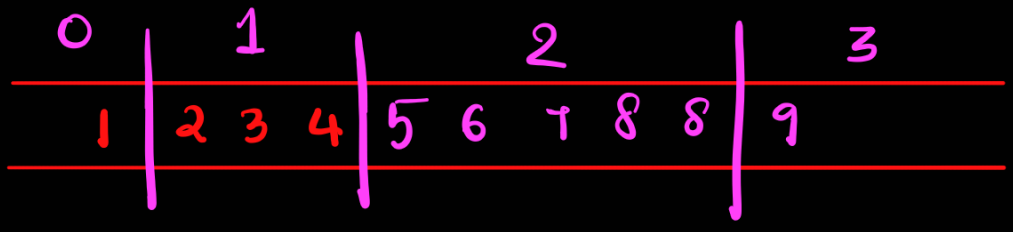
Relaxing edge



Can reach v in 1 step from cur

queue → priority queue

+1 → weight of edge



There are no negative numbers

cannot find distance < something we have seen before

It explores layer by layer

There will not be a layer L_i where a node from L_i goes to L_j $(j<i)$ and it finds lesser distance

Finds distance from starting pt to every other node

Implicit graph fn → nbr code changes

```cpp
int n,m;
vector<string> arr;

void solve(){
    cin>>n>>m;
    arr.resize(n);
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
}
```

```cpp
using state = pair<int,int>;

int n,m;
vector<string> arr;

void solve(){
    cin>>n>>m;
    arr.resize(n);
    state st,en;
    for(int i=0;i<n;i++){
        cin>>arr[i];
        for(int j=0;j<m;j++){
            if(arr[i][j]=='A'){
                st = {i,j};
            }else if(arr[i][j]=='B'){
                en = {i,j};
            }
        }
    }
```

```cpp
vector<vector<int>> vis, dist;
void bfs(state st){


}
```

```cpp
const int INF = 1e9;
#define F first
#define S second

using state = pair<int,int>;

int n,m;
vector<string> arr;

vector<vector<int>> vis, dist;
void bfs(state st){
    vis = vector<vector<int>>(n,vector<int>(m,0));
    dist = vector<vector<int>>(n,vector<int>(m,INF));
    queue<state> q;

    dist[st.F][st.S] = 0;
    q.push(st);
```

```cpp
void bfs(state st){
    vis = vector<vector<int>>(n,vector<int>(m,0));
    dist = vector<vector<int>>(n,vector<int>(m,INF));
    queue<state> q;

    dist[st.F][st.S] = 0;
    q.push(st);

    while(!q.empty()){
        state cur = q.front();
        q.pop();
        if(vis[cur.F][cur.S])continue;
        vis[cur.F][cur.S]=1;
        // process the node
        for(auto v:neighbour(cur)){
            // relaxing edge.
            if(dist[v.F][v.S] > dist[cur.F][cur.S]+1){
                dist[v.F][v.S] = dist[cur.F][cur.S]+1;
                q.push(v);
            }
        }
    }

}
```

```cpp
int dx[] = {1,0,-1,0};
int dy[] = {0,-1,0,1};

vector<state> neighbour(state s){


}
```

```cpp
bool inside(int x,int y){
    if(x>=0 && x<n && y>=0 && y<m && arr[x][y]!='#')return 1;
    return 0;
}
```

```cpp
vector<state> neighbour(state s){
    vector<state> ans;
    for(k=0;k<4;k++){
        state temp = {s.F+dx[k],s.S+dy[k]};
        if(inside(temp.F,temp.S)){
            ans.push_back(temp);
        }
    }
    return ans;
}
```

use make_pair

int

```cpp
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            cout<<dis[i][j]<<"\t";
        }
        cout<<endl;
    }
}
```

Input

```
5 8
########
#.A#...#
#.##.#B#
#......#
########
```

Output

```
100  100  100  100  100  100  100  100
100  1    0    100  8    9    10   100
100  2    100  100  7    100  9    100
100  3    4    5    6    7    8    100
100  100  100  100  100  100  100  100
```

Same heuristic visited

Every node in grid is processed exactly once

↓

going through 4 neighbours

$O(N * M * 4) \approx O(N * M)$

```
vector<vector<int>> vis, dist;
vector<vector<state>> par;

void bfs(state st){
    vis = vector<vector<int>>(n,vector<int>(m,0));
    dist = vector<vector<int>>(n,vector<int>(m,INF));
    par = vector<vector<int>>(n,vector<int>(m,{-1,-1}));
    queue<state> q;
```

state state make_pair

```
// process the node
for(auto v:neighbour(cur)){
    // relaxing edge.
    if(dist[v.F][v.S] > dist[cur.F][cur.S]+1){
        dist[v.F][v.S] = dist[cur.F][cur.S]+1;
        par[v.F][v.S] = cur;
        q.push(v);
    }
}
```

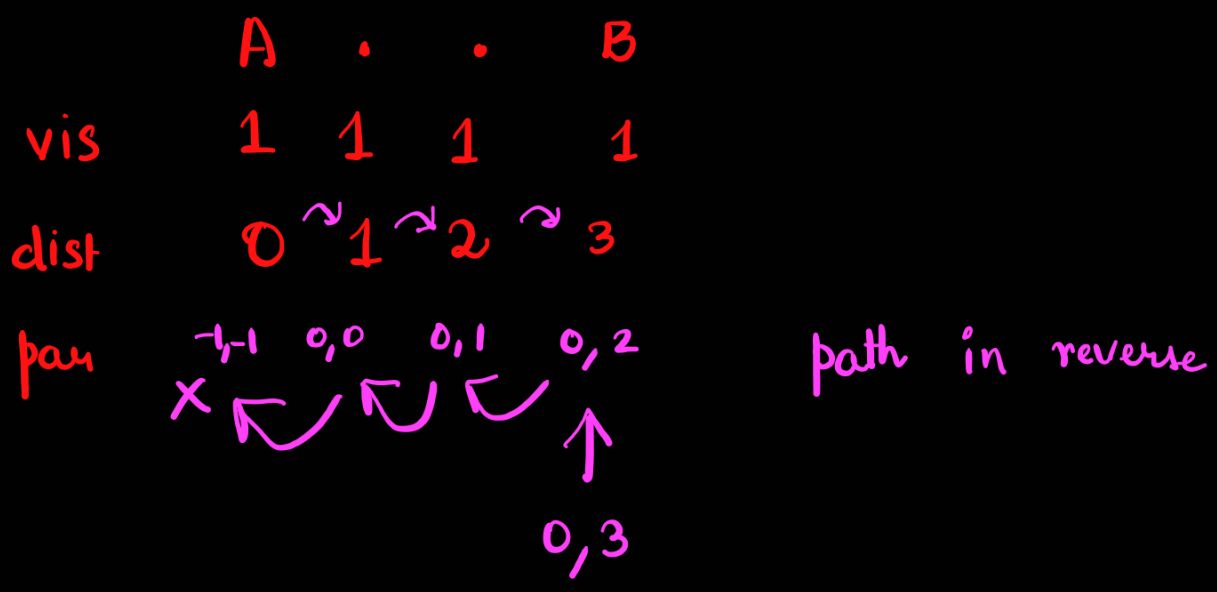
```
bfs(st);
if(vis[en.F][en.S]){
    cout<<"YES\n";
    cout<<dist[en.F][en.S]<<endl;

    vector<state> path;
    state cur = en;
    while(cur!={-1,-1}){
        path.push_back(cur);
        cur = par[cur.F][cur.S];
    }
    for(auto v:path){
        cout<<v.F<<" "<<v.S<<endl;
    }

}else{
    cout<<"NO\n";
}
```

state ({-1, -1})

reverse (path); → Now in forward dirn



par - How did we reach SP?
Back pointer