**=> Shortest path to Reach Destination in a Maze with Obstacles:**

Given a MxN matrix where each element can either be 0 or 1. We need to find the **shortest path** between a given source cell to a destination cell **& print Path.** The path can only be created out of a cell if its value is 1.

Expected time complexity is O(MN).

For example –

**Input:**

mat[ROW][COL] = {{**1**, 0, **1**, **1**, **1**, 1, 0, 1, 1, 1 },

{**1**, 0, **1**, 0, **1**, 1, 1, 0, 1, 1 },

{**1**, **1**, **1**, 0, **1**, 1, 0, 1, 0, 1 },

{0, 0, 0, 0, **1**, 0, 0, 0, 0, 1 },

{1, 1, 1, 0, 1, 1, 1, 0, 1, 0 },

{1, 0, 1, 1, 1, 1, 0, 1, 0, 0 },

{1, 0, 0, 0, 0, 0, 0, 0, 0, 1 },

{1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },

{1, 1, 0, 0, 0, 0, 1, 0, 0, 1 }};

Source = {0, 0}; Destination = {3, 4};

**Output:**

11

[0,0]

[1,0]

[2,0]

[2,1]

[2,2]

[1,2]

[0,2]

[0,3]

[0,4]

[1,4]

[2,4]

[3,4]

=> **SOLn:**

**=> BFS first visits all Cells at distance ‘1’ from Source , then at distance ‘2’ and so on, so the first time we reach the destination , we are guaranteed that it’s the Shortest path.**

**=> USE :**

**//LEFT , DOWN , UP , RIGHT**

int rowNum[] ={-1,0,0,1};

int colNum[] ={0,-1,1,0};

The idea is inspired from [Lee algorithm](https://en.wikipedia.org/wiki/Lee_algorithm) and uses BFS.

1. We start from the source cell and call BFS procedure.
2. We maintain a queue to store the coordinates of the matrix and initialize it with the source cell.
3. We also maintain a Boolean array visited of same size as our input matrix and initialize all its elements to false.
   1. We LOOP till queue is not empty
   2. Dequeue front cell from the queue
   3. Return if the destination coordinates have reached.
   4. **For each of its four adjacent cells, if the value is 1 and they are not visited yet, we enqueue it in the queue and also mark them as visited,and store current node as its parent .**

**=> CODE :**

#include <bits/stdc++.h>

using namespace std;

#define ROW 9

#define COL 10

struct queueNode{

pair<int,int> pt; **//Coordinates of cell.**

int dist; **//Cell's distance from the source.**

};

**//LEFT , DOWN , UP , RIGHT**

int rowNum[] ={-1,0,0,1};

int colNum[] ={0,-1,1,0};

**//Check whether we are not going out of current matrix.**

bool isValid(int row,int col)

{

return(row>=0 && row<ROW

&& col>=0 && col<COL);

}

**//Store Parent of Each Cell.**

pair<int,int> parent[ROW][COL];

**//If a Cell is not visited, its parent= {-1,-1}.**

void initializeParent()

{

for(int i=0;i<ROW;i++)

{

for(int j=0;j<COL;j++)

{

parent[i][j]={-1,-1};

}

}

}

int BFS(int mat[][COL],pair<int,int> src, pair<int,int> dest)

{

**//In any pair, 'x' coordinate = first, second= 'y' coordinate;**

**//Source & Dest should not be Blocked.**

if(!mat[src.first][src.second] || !mat[dest.first][dest.second]) return -1;

**//Set up for BFS.**

bool visited[ROW][COL]={0};

visited[src.first][src.second]=true;

queueNode s={src,0};

queue<queueNode> q;

q.push(s);

while(!q.empty())

{

queueNode curr = q.front();

q.pop();

pair<int,int> pt = curr.pt; //IMP.

**//Check whether we reached the destination.**

if(pt.first==dest.first && pt.second==dest.second)

return curr.dist;

**//Enqueue the Adjacent cells into the queue.**

for(int i=0;i<4;i++)

{

int row = pt.first + rowNum[i];

int col = pt.second + colNum[i];

**//If the cell we are checking is not out of the matrix**

**// AND it is not BLOCKED AND it's not visited ,**

**//then Mark it Visited ,store it's parent & enqueue.**

if(isValid(row,col) && mat[row][col] && !visited[row][col])

{

visited[row][col]=true;

parent[row][col]={pt.first,pt.second};

queueNode adjCell ={ {row,col}, curr.dist+1 };

q.push(adjCell);

}

}

}

**//BFS above will visit all Cells in Worst Case and if Execution reached**

**//till here means there is no path to Destination , return -1;**

return -1;

}

int main(){

**//Initialize the Matrix.**

int mat[ROW][COL] =

{

{ 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },

{ 1, 0, 1, 0, 1, 1, 1, 0, 1, 1 },

{ 1, 1, 1, 0, 1, 1, 0, 1, 0, 1 },

{ 0, 0, 0, 0, 1, 0, 0, 0, 0, 1 },

{ 1, 1, 1, 0, 1, 1, 1, 0, 1, 0 },

{ 1, 0, 1, 1, 1, 1, 0, 1, 0, 0 },

{ 1, 0, 0, 0, 0, 0, 0, 0, 0, 1 },

{ 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },

{ 1, 1, 0, 0, 0, 0, 1, 0, 0, 1 }

};

initializeParent();

pair<int,int> src = {0 , 0} , dest = { 3,4 };

int dist = BFS(mat ,src , dest);

cout<<dist<<endl; //Returns (-1) if no path to dest.

**//Code to Print Shortest Path if it exists.**

if(dist!=-1){

list<pair<int,int>> path;

pair<int,int> curr = dest;

while(curr.first!=-1 && curr.second!=-1)

{

path.push\_front(curr);

curr = parent[curr.first][curr.second];

}

for(auto it:path){

cout<<"["<<it.first<<","<<it.second<<"]"<<endl;

}

}

}

**-> Time = O(V+E) for BFS and**

**-> Space = O(V+E) because of queue,visited array,parent array**