Graphs

Content

1. Coloring A Border

Problem

You are given an m x n integer matrix grid, and three integers row, col, and color. Each value in the grid represents the color of the grid square at that location.Two squares belong to the same connected component if they have the same color and are next to each other in any of the 4 directions. The border of a connected component is all the squares in the connected component that are either 4-directionally adjacent to a square not in the component, or on the boundary of the grid (the first or last row or column). You should color the border of the connected component that contains the square grid[row][col] with color. Return the final grid.

Approach

The primary intuition is to do a DFS from the starting cell and find all the cells of the oldColor that needs to be changed. We mark these cells with a negative value of the oldColor. Once this is done, we need to find out which among those cells lies interior and which lies exterior. Interior cells have all 4 neighboring cells(top, bottom, left and right) to have either the oldColor value or -oldColor value. Make these interior cells positive again. Once we have processed this for all necessary nodes from the starting cell, we will get a grid containing negative cells that denote the boundary. We need to sweep through the entire grid and change these negative values to the new color.

Steps

1. Check for existing null or empty grid and return null if so.
2. Store the color of the starting cell grid[r0][c0] in oldColor.
3. Initiate a DFS from the starting cell.

4. Check if the current cell lies out of bounds off the grid or if the current cell does not have the same color as the starting cell and return if so.
5. Otherwise, change the current cell's color to a negative value for us to remember that we have already visited the cell
6. Do a DFS for all neighbouring points that are up, down, left & right from the current cell.
7. Once DFS returns back for the current cell after processing all directions from it, change the current cell's color back to positive value if you find that the current cell lies within adjacent cells top, bottom, left & right with the same value.
8. Once the entire DFS has been done, we now have a grid containing negative values representing the border which needs to be recolored to the new color.

Implementation

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/coloring_a_border.cpp

Zero One matrix

Problem

Given an `m x n` binary matrix `mat`, return the distance of the nearest `0` for each cell. The distance between two adjacent cells is `1`.

Approach

We know the zero value cell has the minimum distance 0, then we can spead from these zero cells step by step to get the minimum distance between zero cells with one cells. So we use a bfs solution.

First, we push the zero cells into the queue. For each step, we pop one cell from the queue and push its unvisited neighbors into the queue and update the distance.

Implementation

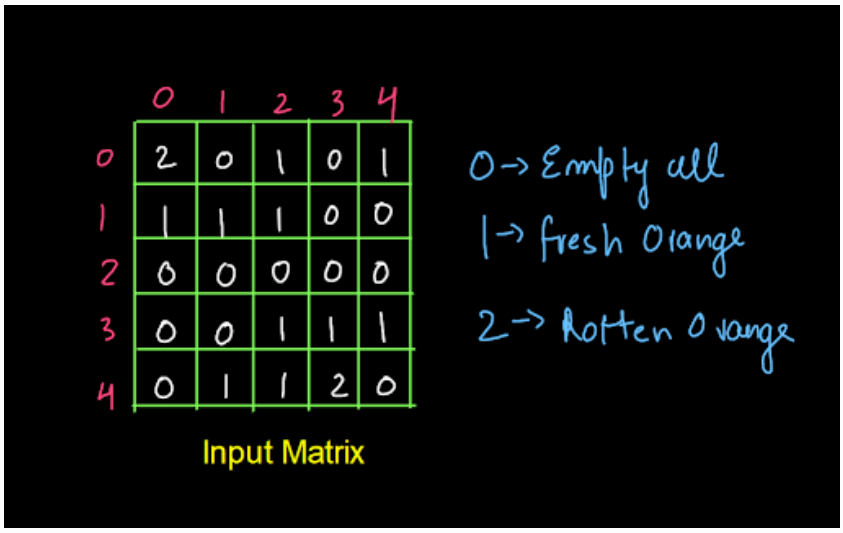
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/zeroOneMatrix.cpp

Time complexity - O(mn)

Rotting Oranges

Problem

We are given an M x N matrix filled with 0's, 1's and 2's. In this matrix, the 0's represent an empty cell i.e. nothing is present in this cell. The 1's represent fresh oranges and the 2's represent rotten oranges. Every cell is adjacent to 4 other cells and they are top, bottom, left and right to that cell.



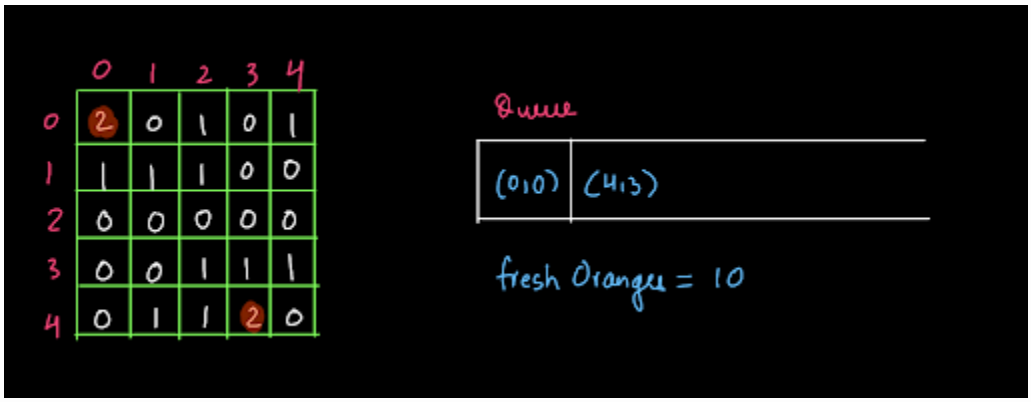
If a rotten orange takes 1 hour to rotten all the fresh oranges adjacent to it, we have to tell that in how many hours all the oranges in the matrix will be rotten. If there are one or more oranges that can't be rotten by the adjacent oranges, return -1.

Approach

This is a pretty basic problem if you think about it carefully. We can solve this problem easily using BFS i.e. Breadth First Traversal of a graph.

What and Why of the problem

We will use a Linked List or we may also use a queue. Have a look at the diagram shown below:



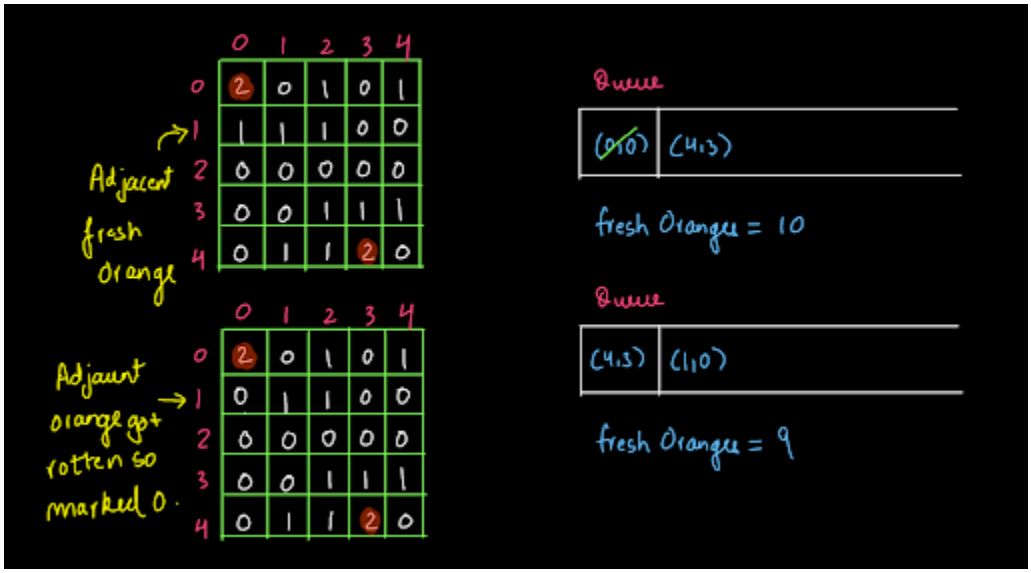
We have filled the queue with the positions where we have the rotten oranges in the matrix (where we have value 2 in the matrix). This is the initial step that we have done. Also, we have counted the number of fresh oranges initially in the matrix. Why have we counted it? Think!!!

Now, we will start with the repeating steps. First, we will determine the size of the queue. This will tell us the number of times we have to do the "remove-work-add" procedure that we learnt in Level-order traversal and when this inner loop will be complete, we would have completed one level.

Does it sound confusing to you? It should not if you have already studied the level-order line-wise in depth in the foundation course.

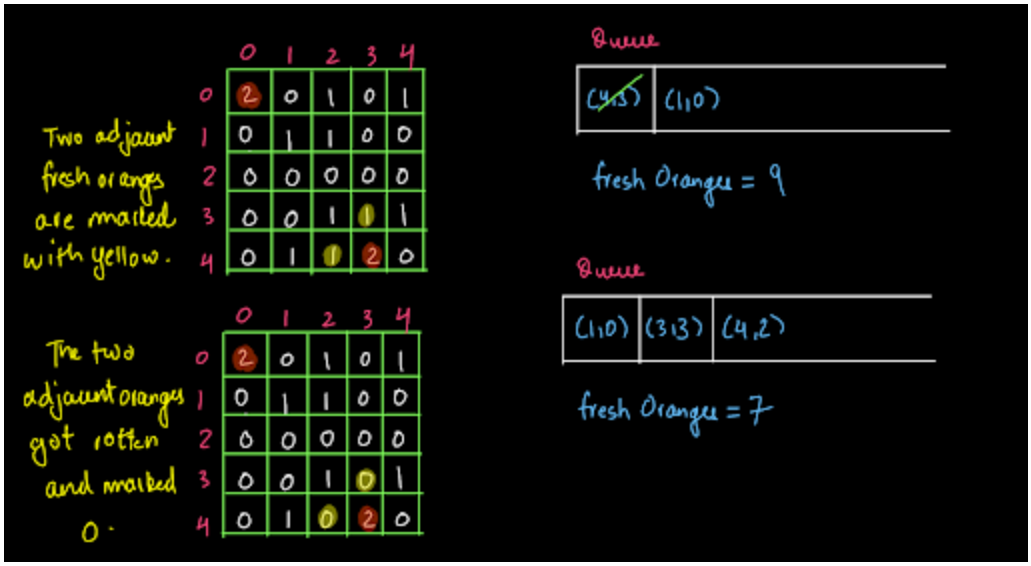
So, let's start our procedure. As we can see in the matrix given to us, there are initially two rotten oranges and they are already kept inside the queue. So, the first level has two child nodes (level order of a tree).

Have a look at the image shown below:



So, what have we done here? We have followed the "remove-work-add" algorithm. We removed the first rotten orange and added the fresh orange that was its neighbor into the queue. Also, after adding the fresh orange to the queue, it should not be marked fresh in the matrix. So, we can either mark it as 2 i.e. rotten or we can mark it as 0. We prefer to mark the oranges added to the queue as 0 as the 0 valued cell is never touched in our entire process. So, it is kind of a safe thing to do so. Also, when we add the adjacent fresh oranges to the queue depicting that they will get rotten now, the count of the fresh oranges that we maintained at the beginning will be reduced.

We are not finished with the procedure here. We counted the size of the queue as 2 initially. So, let us again repeat this procedure of "remove-work-add" for the next element present at the front of the queue.



So, after this, we have completed one level or we may say that 1 hour has passed and few oranges have rotten. Now, we will again repeat the same procedure i.e. calculate the size of the queue and apply our "remove-work-add" algorithm to those many number of elements.

This procedure will continue till the queue becomes empty. After the queue becomes empty, we will check the count of the fresh oranges. If the count of the fresh oranges is 0, this means that all the oranges were rotten and the time taken will be given by the number of levels in the level-order traversal or BFS of the graph.

If the count of fresh oranges is not 0, this means that we have that many oranges that cannot be rotten by any orange. So, we will return -1.

We recommend you refer to the solution video () to understand the complete procedure and get a dry run of it also. Now that we have completely understood the procedure, let us write the code for the same.

Time Complexity

The time complexity of the above code is O(M x N) where M is the number of rows and N is the number of columns of the matrix. This is because first we have

to traverse the entire matrix to get the count of fresh oranges and the rotten oranges too and later we apply BFS on all the rotten oranges.

Space Complexity

The space complexity will also be O(M x N) as we have made a queue/ Linked List in which we add all the rotten oranges and the worst case scenario can be that all the oranges are rotten i.e. M x N and such is the average case complexity as well.

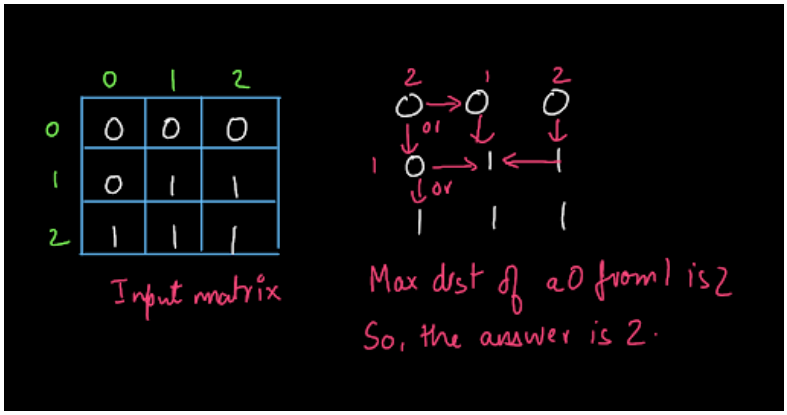
Implementation

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/rottingOranges.cpp

As far as land as possible

Problem

We will be given a grid/matrix of size N x N. It will contain only 0's and 1's. The 0's represent water and the 1's represent the land. We have to calculate the maximum nearest distance of water from the land. What do we mean by this? Have a look at the diagram given below:



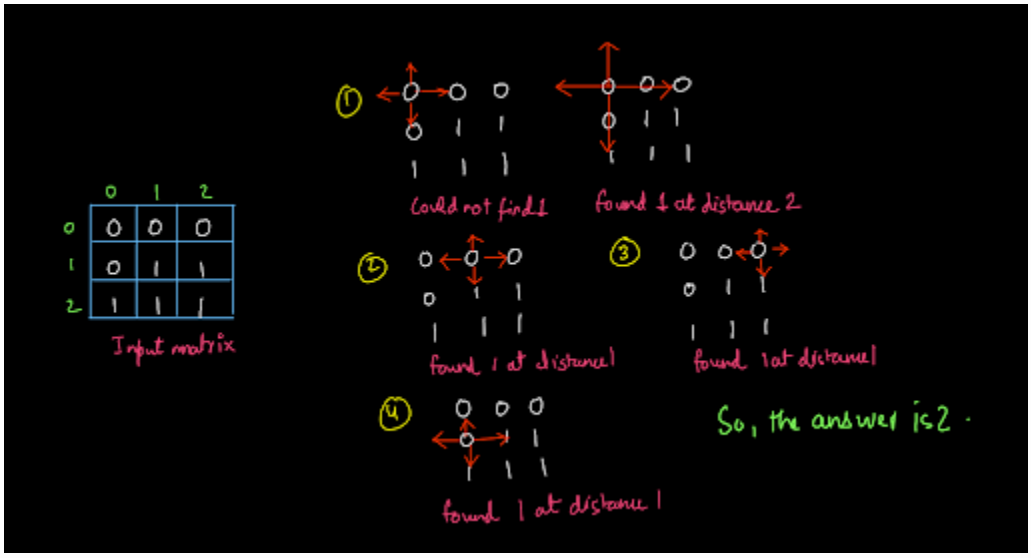
We have calculated the distance of nearest land from water i.e. the nearest distance of 1's from 0's. Now, we have to return the maximum of all the distance values that we have calculated showing that at that cell, the water is the farthest from land or as far from land as possible. We recommend you refer to the problem video to understand the question completely.

Approach

So, let us first discuss a brute force approach that we can follow and then we will move to the optimized approach.

Brute Force Approach

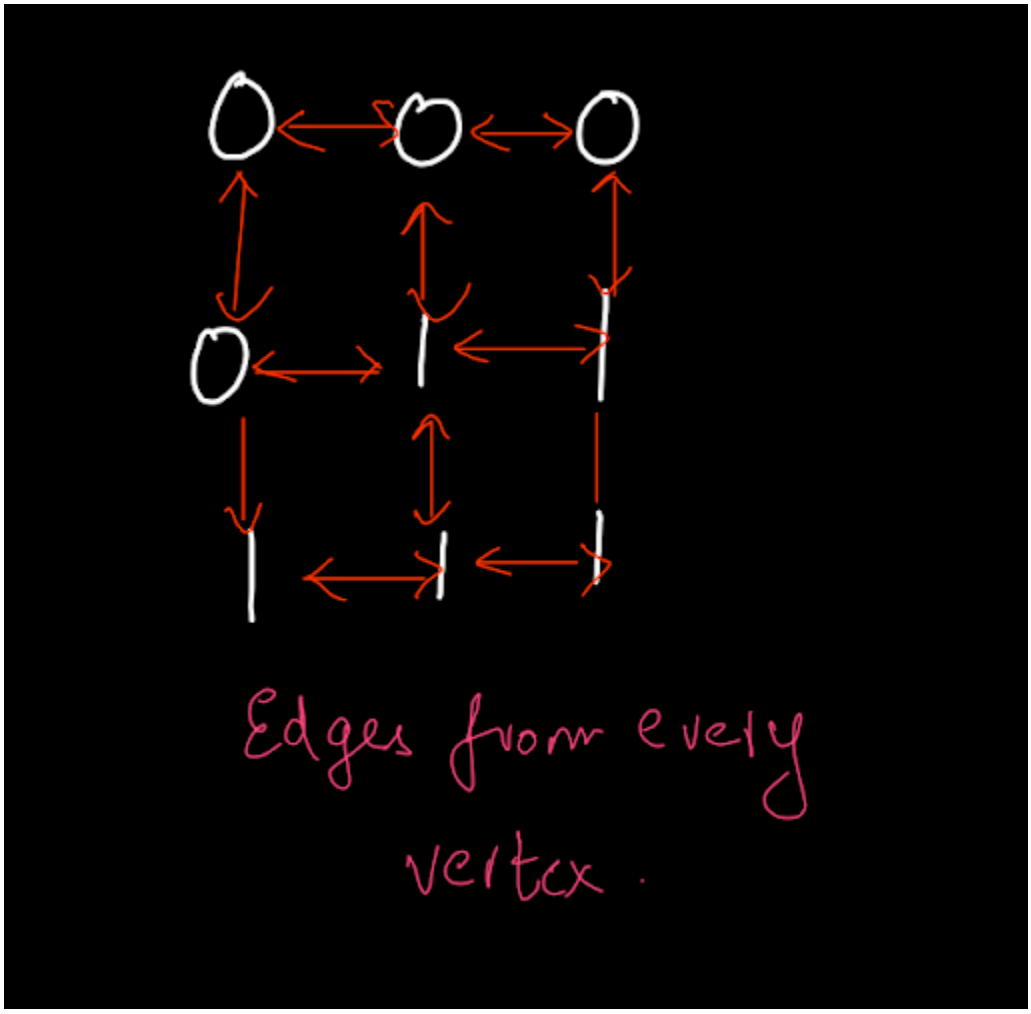
If we understand the main point of the question, it is just to calculate the distance of 1's from 0's and then return the maximum out of them. So, at every 0 in the matrix, we can apply BFS and find the nearest 1 to it. For instance, have a look at the image shown below:



So, as you can see from the image above, applying BFS on every 0 in the matrix will give us the nearest distance of a 1 from it. This is to be done for every 0 in the matrix. We are not going to code this approach as you already know how to apply BFS. Let us discuss the complexity of this approach once.

Time Complexity of the Brute Force Approach:

See, we are applying the BFS on every 0. So, the time complexity will be dependent on the number of 0s in the matrix. Let us say that the number of zeroes is M. Also, the time taken by BFS is O(V+E) where V is the number of vertices in the graph and E is the number of edges. In the given matrix, every cell is a vertex. Hence, there are $O(n^2)$ vertices. Also, have a look at the image shown below:



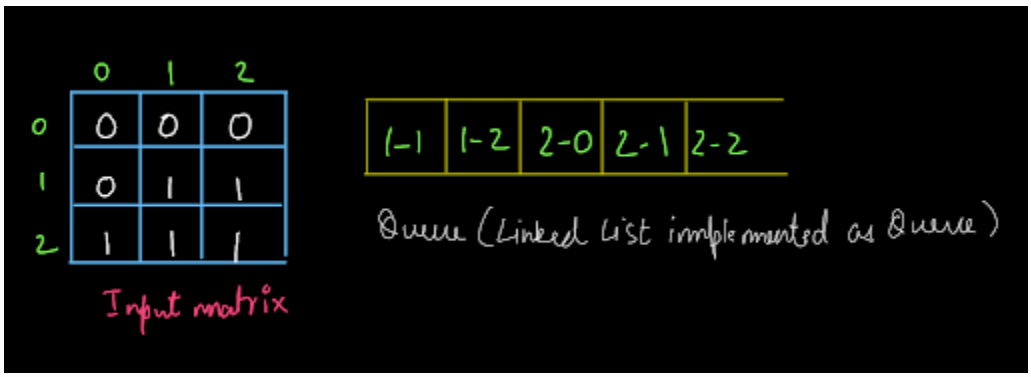
So, the image shows that almost 4 edges are there corresponding to every vertex. They are not exactly 4 but still there are almost 4 vertices for every edge hence the number of vertices will be approximately $4n^2$. So, $E + V = n^2 + 4n^2 = O(n^2)$. Hence the time complexity will be $O(M \times n^2)$ where M is the number of zeroes.

Optimized Approach

We will now improve the above complexity in such a way that the complexity will become $O(n^2)$ and will not depend on the number of zeroes.

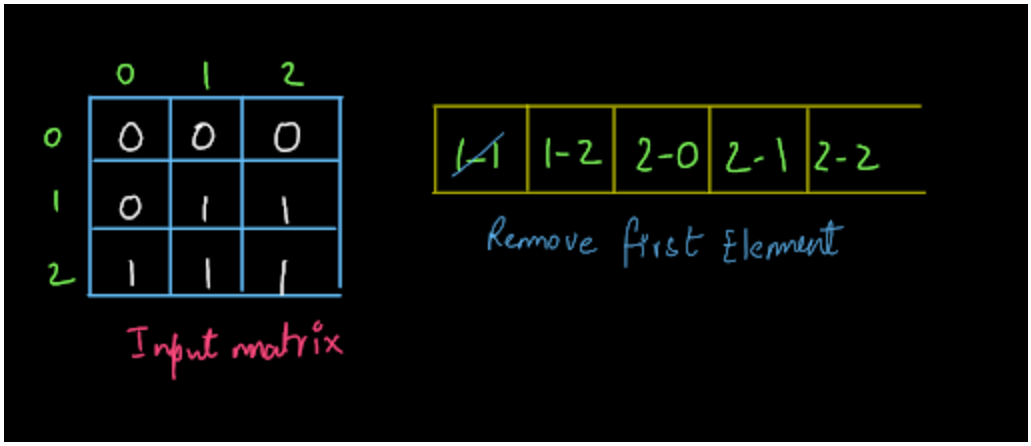
We recommend you refer to the Zero-One Matrix problem as we are going to discuss the same procedure and it will be really helpful if you have already studied it.

So, instead of applying BFS on each 0 in the matrix, we will apply BFS on the 1's present in the matrix simultaneously. This is called as multi-source BFS, Have a look at the image shown below:

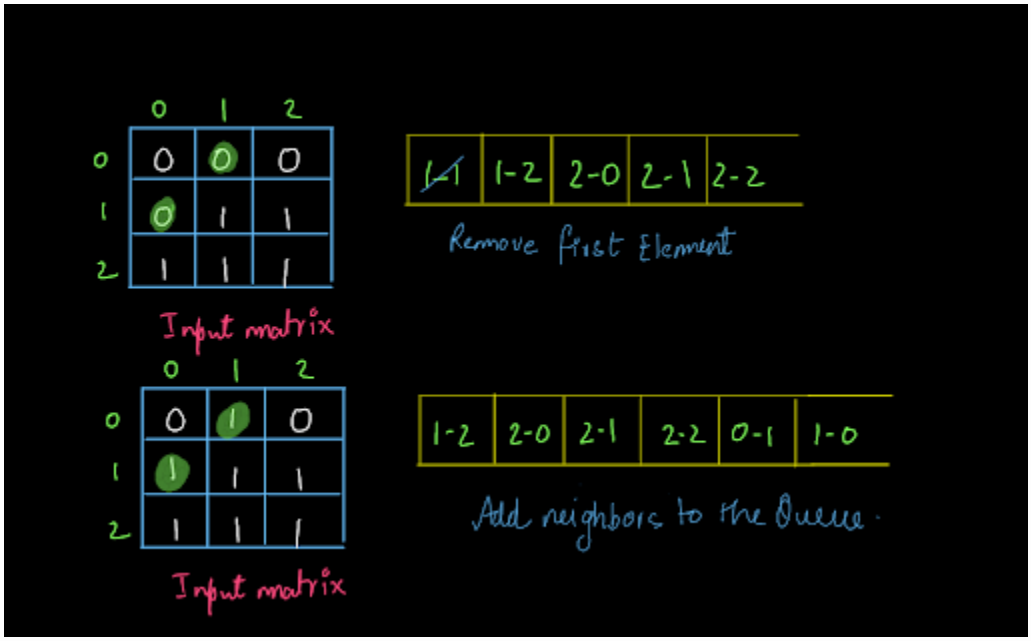


As shown above, we will take a queue and initially we will store the positions of all the 1's present in the matrix. Now, as the Level-order Traversal of a tree, we will count the number of elements present in the queue and this will be the number of nodes in the first level of the tree.

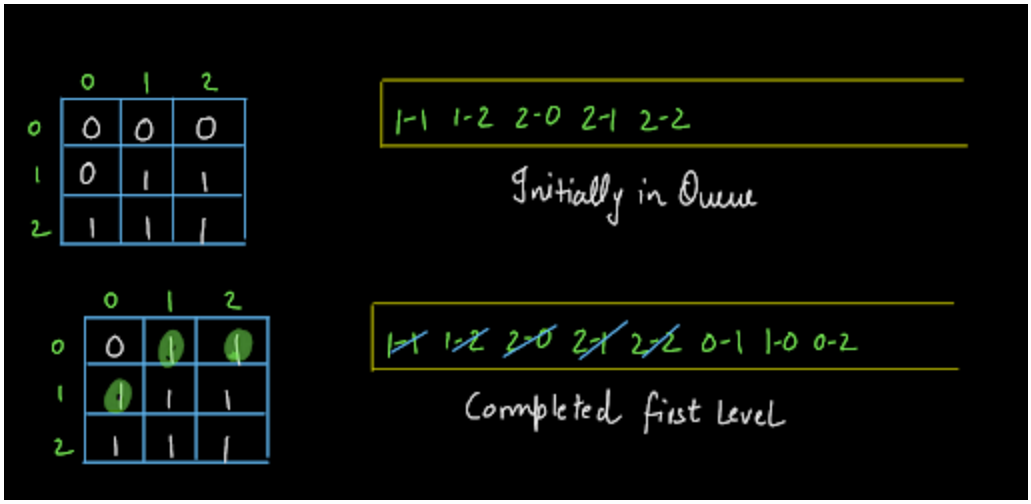
We will apply the "remove-work-add" algorithm that we studied in the Level-order Traversal of Generic Tree and we will remove the first element from the queue as shown below:



After removing the first element, we find whether there is a 0 adjacent (neighbor) to the current removed element or not. If there is any 0, the "work" will be to mark it visited by storing 1 at that position and we will "add" its position into the queue.



Now, we will continue with the same "remove-work-add" algorithm for the next elements present in the queue.



Since initially, we counted 4 as the size of the queue, after we removed the first 4 elements of the queue, we will move to the next level in the tree and this next level indicates the distance of 1 from 0.

So, we will keep on continuing this procedure till the queue becomes empty and the last level at which we found a 0 will be the answer to our problem.

Implementation

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/asFarAsLandAsPossible.cpp