MEDIUM

Shortest Distance in a Binary Maze

Intuition

Given : Matrix, we have to move using the shortest possible path in the binary maze

Eg.

1	**1**	1	1
1	1	0	1
1	1	**1**	1
1	1	0	0
1	0	0	0

Bold - source and destination

Source = [0,1]
Destination = [2,2]

Movement is allowed only in 4-adjacent directions

Eg.

1	**1**	1	1
1	1	0	1
1	1	**1**	1
1	1	0	0
1	0	0	0

6 Steps
3 Steps
5 Steps
So we will return 3 as the answer

We can apply either the Dijkstra's Algorithm using the :

- Queue
- Set Data Structure

Priority Queue

We will take a priority queue and a 2D array to store all the distances in dijkstra but We started with a cell 0 in all 3 possible directions with a distance of 1, we used priority queue was because we wanted minimal of them but here we have all having same distance ie. It can be solved using a normal queue.

we can now see that those are already stored in an increasing fashion ie. 1 -> 2 -> 3 … as the distance is increasing in a uniform fashion so we need only a normal queue so we do not need addition log(N) complexity

Eg.

Distance Vector // initial configuration

inf	0	inf	inf
inf	inf	inf	inf
inf	inf	inf	inf
inf	inf	inf	inf
inf	inf	inf	inf

Queue

[0, [0,1]] // queue

Traversal using Dijkastra's Algorithm

Distance Vector // initial configuration

1	**0**	1	2
2	**1**	2	3
3	**2**	**3**	4
4	3	inf	inf
inf	inf	inf	inf

Queue

[0, [0,1]]

[1,[0,2]]
[1,[0,0]]
[1,[1,1]]

[2,[0,3]]
[1,[0,0]]
[1,[0,2]]

[2,[2,1]]
[2,[0,3]]
[1,[0,0]]

[2,[1,0]]
[2,[2,1]]
[2,[0,3]]

[3,[1,3]]
[2,[1,0]]
[2,[2,1]]

[3,[2,2]] // Destination - Break and return distance = 3
[3,[1,3]]
[2,[1,0]]

Approach

- Checking the edge case if we are already at the destination :
    - Return 0
- Calculate the dimensions of the grid
- Create a distance vector and initialize with infinity to all elements
- Create a queue to store the { distance, {row, col}}
- Insert the source into the queue with distance 0
- Mark the distance of the source as 0
- Traverse until the queue becomes empty :
    - Extract the first element of the queue
    - Check if the element is our destination :

        - Return the distance to reach node
    - Traverse for all of the adjacent components of the node :
        - Calculate the index of the adjacent elements
        - Check for the validity of the adjacent node :
            - Check if the adjacent node is 1 and the distance of node + 1 is smaller than the distance to node :
                - Update the distance to the adjacent node as distance of node +1
                - Push the adjacent node into the queue with updated distance
- Return -1 // as in this case we will never be able to reach the destination

Function Code

```
int shortestPath(vector<vector<int>> &grid, pair<int, int> source,
                     pair<int, int> destination) {

        // checking if we are already standing at the destination then we
need to return distance 0
        if(source.first==destination.first &&
source.second==destination.second)
        {
            return 0;
        }

        // calculating the dimensions of the grid
        int n = grid.size();
        int m = grid[0].size();
        // Declaring the distance vector that contains inf in start
        vector<vector<int>> distance(n,vector<int>(m,1e9));
        // creating a queue to store the final elements
        queue<pair<int,pair<int, int>>> q;
        // inserting the source element with 0 distance into the queue
        q.push({0,{source.first,source.second}});
        distance[source.first][source.second] = 0;
        // traversing until the queue becomes empty
        while(!q.empty())
        {
            // extracting the first element from the queue
            int dist = q.front().first;
            int row = q.front().second.first;
            int col = q.front().second.second;
            // popping the first element
```

```
            q.pop();
            // checking if the element is our destination
            if(row==destination.first && col==destination.second)
                {
                    return dist;
                }

            // traversing for the adjacent components
            int delRow[] = {-1,0,1,0};
            int delCol[] = {0,1,0,-1};

            for(int i=0;i<4;i++)
            {
                // calculating the dimensions of the neighbor
                int nrow = row+delRow[i];
                int ncol = col+delCol[i];

                // checking for validity
                if(nrow<n && ncol<m && nrow>=0 && ncol>=0)
                {
                    // checking if the node can be reached with a better
distance
                    if(dist+1<distance[nrow][ncol] && grid[nrow][ncol]==1)
                    {
                        // updating the distance and pushing the element
into the queue
                        distance[nrow][ncol] = dist+1;

                        q.push({dist+1,{nrow,ncol}});
                    }
                }
            }
        }
        return -1;
    }
```

Time Complexity

O(n * m)