January 5, 2023 ▪ Data Structure / Graph

G-36: Shortest Distance in a Binary Maze

Problem Statement:

Given an **n * m** matrix grid where each element can either be **0** or **1.** You need to find the shortest distance between a given source cell to a destination cell. The path can only be created out of a cell if its value is 1.

If the path is not possible between the source cell and the destination cell, then return **-1**.

Note: You can move into an adjacent cell if that adjacent cell is filled with element 1. Two cells are adjacent if they share a side. In other words, you can move in one of four directions, Up, Down, Left, and Right.

Examples:

```
Example 1:

Input:
grid[][] = {{1, 1, 1, 1},
            {1, 1, 0, 1},
            {1, 1, 1, 1},
            {1, 1, 0, 0},
            {1, 0, 0, 1}}
source = {0, 1}
destination = {2, 2}
Output:
3

Explanation:

1 1 1 1
1 1 0 1
1 1 1 1
1 1 0 0
1 0 0 1
The highlighted part in the
above matrix denotes the
shortest path from source to
destination cell.

Example 2:

Input:
grid[][] = {{1, 1, 1, 1, 1},
            {1, 1, 1, 1, 1},
            {1, 1, 1, 1, 0},
            {1, 0, 1, 0, 1}}
source = {0, 0}
destination = {3, 4}
Output:
-1
Explanation:
Since, there is no path possible
between the source cell and the
destination cell, hence we
return -1.
```

Recent Posts

Implement Upper Bound

Implement Lower Bound

2023 – Striver's SDE Sheet Challenge

Top Array Interview Questions – Structured Path with Video Solutions

Longest Subarray with sum K | [Postives and Negatives]

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Problem Link

Note: In case any image/dry run is not clear please refer to the video attached at the bottom.

Approach:

We'll solve this problem by Dijkstra's Algorithm using a simple queue. Since, there is no adjacency list for this particular problem we can say that the adjacent cell for a coordinate is that cell which is either on the top, bottom, left, or right of the current cell i.e, a cell can have a maximum of 4 cells adjacent to it.

Initial configuration:

- **Source Node and Destination Node:** Before starting off with the Algorithm, we need to define a source node and a destination node, between which we need the shortest possible distance.
- **Queue:** Define a Queue which would contain pairs of the type {dist, pair of coordinates of cell }, where 'dist' indicates the currently updated value of the shortest distance from the source to the cell.
- **Distance Matrix:** Define a distance matrix that would contain the distance from the source cell to that particular cell. If a cell is marked as 'infinity' then it is treated as unreachable/unvisited.

The Algorithm consists of the following steps :

- Start by creating a queue that stores the distance-node pairs in the form {dist, coordinates of cell pair} and a dist matrix with each cell initialized with a very large number (to indicate that they're unvisited initially) and the source cell marked as '0'.
- We push the source cell to the queue along with its distance which is also 0.
- Pop the element at the front of the queue and look out for its adjacent nodes (left, right, bottom, and top cell). Also, for each cell, check the validity of the cell if it lies within the limits of the matrix or not.

- If the current reachable distance to a cell from the source is better than the previous distance indicated by the distance matrix, we update the distance and push it into the queue along with cell coordinates.
- A cell with a lower distance would be at the front of the queue as opposed to a node with a higher distance. We repeat the above two steps until the queue becomes empty or until we encounter the destination node.
- Return the calculated distance and stop the algorithm from reaching the destination node. If the queue becomes empty and we don't encounter the destination node, return '-1' indicating there's no path from source to destination.
- Here's a quick demonstration of the algorithm :

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Intuition: Here in this problem, instead of a graph we have a 2D binary matrix in which we have to reach a destination cell from a source cell. So, we can see that this problem is easily approachable by Dijkstra's Algorithm. Now, here we use a **queue** instead of a **priority queue** for storing the distance-node pairs. Let's understand through an illustration why a queue is better here:

We can see clearly in the above illustration that the distances are increasing monotonically (because of constant edge weights). Since greater distance comes at the top automatically, so we do not need the priority queue as the pop operation will always pop the smaller distance which is at the front of the queue. This helps us to eliminate an additional log(N) of time needed to perform insertion-deletion operations in a priority queue.

Code:

C++ Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    int shortestPath(vector<vector<i
                     pair<int, int>
    {
        // Edge Case: if the source
        if (source.first == destinat
            source.second == destina
            return 0;

        // Create a queue for storin
        // in the form {dist,{cell c
        queue<pair<int, pair<int, in
        int n = grid.size();
        int m = grid[0].size();

        // Create a distance matrix
        // unvisited and the source
        vector<vector<int>> dist(n,
        dist[source.first][source.se
        q.push({0, {source.first, so
```

```
            // The following delta rows
            // each index represents eac
            // in a direction.
            int dr[] = {-1, 0, 1, 0};
            int dc[] = {0, 1, 0, -1};

            // Iterate through the maze
            // and pushing whenever a sh
            while (!q.empty())
            {
                auto it = q.front();
                q.pop();
                int dis = it.first;
                int r = it.second.first;
                int c = it.second.second

                // Through this loop, we
                // for a shorter path to
                for (int i = 0; i < 4; i
                {
                    int newr = r + dr[i]
                    int newc = c + dc[i]

                    // Checking the vali
                    if (newr >= 0 && new
                    == 1 && dis + 1 < di
                    {
                        dist[newr][newc]

                        // Return the di
                        // we encounter
                        if (newr == dest
                            newc == dest
                            return dis +
                        q.push({1 + dis,
                    }
                }
            }
            // If no path is found from
            return -1;
        }
    };

    int main()
    {
```

```
    // Driver Code.

    pair<int, int> source, destinati
    source.first = 0;
    source.second = 1;
    destination.first = 2;
    destination.second = 2;

    vector<vector<int>> grid = {{1,
                                 {1,
                                 {1,
                                 {1,
                                 {1,

    Solution obj;

    int res = obj.shortestPath(grid,

    cout << res;
    cout << endl;

    return 0;
}
```

Output :

3

Time Complexity: O(4*N*M) { N*M are the total cells, for each of which we also check 4 adjacent nodes for the shortest path length}, Where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Space Complexity: O(N*M), Where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Java Code

```
import java.util.*;

class tuple {
    int first, second, third;
    tuple(int _first, int _second, i
        this.first = _first;
        this.second = _second;
        this.third = _third;
    }
}
class Solution {

    int shortestPath(int[][] grid, i

        // Edge Case: if the source
        if(source[0] == destination[
           source[1] == destination[

        // Create a queue for storin
        // in the form {dist,{cell c
        Queue<tuple> q = new LinkedL
        int n = grid.length;
        int m = grid[0].length;

        // Create a distance matrix
        // unvisited and the source
        int[][] dist = new int[n][m]
        for(int i = 0;i<n;i++) {
            for(int j =0;j<m;j++) {
                dist[i][j] = (int)(1
            }
        }
        dist[source[0]][source[1]] =
        q.add(new tuple(0, source[0]

        // The following delta rows
        // each index represents eac
        // in a direction.
        int dr[] = {-1, 0, 1, 0};
        int dc[] = {0, 1, 0, -1};

        // Iterate through the maze
        // and pushing whenever a sh
        while(!q.isEmpty()) {
            tuple it = q.peek();
```

```
                q.remove();
                int dis = it.first;
                int r = it.second;
                int c = it.third;

                // Through this loop, we
                // for a shorter path to
                for(int i = 0;i<4;i++) {
                    int newr = r + dr[i]
                    int newc = c + dc[i]

                    // Checking the vali
                    if(newr >= 0 && newr
                    && grid[newr][newc]
                        dist[newr][newc]

                        // Return the di
                        // we encounter
                        if(newr == desti
                           newc == desti
                        q.add(new tuple(
                    }
                }
            }
            // If no path is found from
            return -1;
        }
    }

    class tuf {

        public static void main(String[]

            int[] source={0,1};
            int[] destination={2,2};

            int[][] grid={{1, 1, 1, 1},
                {1, 1, 0, 1},
                {1, 1, 1, 1},
                {1, 1, 0, 0},
                {1, 0, 0, 1}};

            Solution obj = new Solution(
            int res = obj.shortestPath(g

            System.out.print(res);
```

```
            System.out.println();
        }
    }
```

Output :

3

Time Complexity: O(4*N*M) { N*M are the total cells, for each of which we also check 4 adjacent nodes for the shortest path length}, Where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Space Complexity: O(N*M), Where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

> Special thanks to **Priyanshi Goel** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

« Previous Post
Articulation Point in Graph: G-56

Next Post »
G-37: Path With Minimum Effort

Load Comments

takeUforward

The best place to learn data structures, algorithms, most asked coding interview questions, real interview experiences free of cost.

Follow Us

DSA Playlist	DSA Sheets	Contribute
Array Series	Striver's SDE Sheet	Write an Article
Tree Series	Striver's A2Z DSA Sheet	
Graph Series	SDE Core Sheet	
DP Series	Striver's CP Sheet	