January 5, 2023 ▪ Data Structure / Graph

G-37: Path With Minimum Effort

You are a hiker preparing for an upcoming hike. You are given heights, a 2D array of size rows x columns, where heights[row][col] represents the height of the cell (row, col). You are situated in the top-left cell, (0, 0), and you hope to travel to the bottom-right cell, (rows-1, columns-1) (i.e.,**0-indexed**). You can move **up**, **down**, **left**, or **right**, and you wish to find a route that requires the **minimum effort**.

A route's **effort** is the **maximum absolute difference** in heights between two consecutive cells of the route.

Examples:

```
Example 1:
Input:
heights = [[1,2,2],[3,8,2],
```

[5,3,5]]
Output:
2
Explanation:

The route of [1,3,5,3,5] has a maximum absolute difference of 2 in consecutive cells.This is better than the route of [1,2,2,2,5], where the maximum absolute difference is 3.

Example 2:

Input:

heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,1,1,2,1]]
Output:
0
Explanation:
The route of [1,1,1,1,1,1,1,1,1,1,1,1,1,1] has a maximum absolute difference of 0 in consecutive cells.This is better than the route of [1,1,1,1,1,1,2,1], where the maximum absolute difference is 1.

Solution

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

Problem Link

Note: If any image/dry run is unclear, please refer to the video attached at the

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]

bottom.

Approach:

Brute Force: We can figure out the effort for all the paths and return the minimum effort among them.

Optimised (Using Dijkstra) :

In this particular problem, since there is no adjacency list 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 and can only move in these directions.

Initial configuration:

- **Queue:** Define a Queue which would contain pairs of the type {diff, (row, col) }, where 'dist' indicates the currently updated value of difference from source to the cell.

- **Distance Matrix:** Define a distance matrix that would contain the minimum difference 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,(row, col)} 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 difference value of a cell from its parent is better than the previous difference indicated by the distance matrix, we update the difference in the matrix and push it into the queue along with cell coordinates.
- A cell with a lower difference value would be at the front of the queue as opposed to

a node with a higher difference. The only difference between this problem and Dijkstra's Standard problem is that there we used to update the value of the distance of a node from the source and here we update the absolute **difference** of a node from its parent.

- We repeat the above three steps until the queue becomes empty or until we encounter the destination node.
- Return the calculated difference and stop the algorithm from reaching the destination node. If the queue becomes empty and we don't encounter the destination node, return '0' indicating there's no path from source to destination.
- Here's a quick demonstration of the Algorithm's 1st iteration (all the further iterations would be done in a similar way) :

Note: Updating the value of difference will only yield us the **effort** for the path traversed.

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

Intuition:

In this problem, we need to minimize the **effort** of moving from the source cell (0,0) to the destination cell (n – 1,m – 1). The effort can be calculated as the maximum value of the difference between the node and its next node in the path from the source to the destination. Among all the possible paths, we have to **minimize** this effort. So, for these types of minimum path problems, there's one standard algorithm that always comes to our mind and that is Dijkstra's Algorithm which would be used in solving this problem also. We update the distance every time we encounter a value of difference less than the previous value. This way, whenever we reach the destination we finally return the value of difference which is also the **minimum effort.**

Code:

C++ Code

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

class Solution
```

```
{
public:
    int MinimumEffort(vector<vector<
    {

        // Create a priority queue c
        // and their respective dist
        // form {diff, {row of cell,
        priority_queue<pair<int, pai
                       vector<pair<i
                       greater<pair<
            pq;

        int n = heights.size();
        int m = heights[0].size();

        // Create a distance matrix
        // unvisited and the dist fo
        vector<vector<int>> dist(n,
        dist[0][0] = 0;
        pq.push({0, {0, 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 matri
        // and pushing whenever a sh
        while (!pq.empty())
        {
            auto it = pq.top();
            pq.pop();
            int diff = it.first;
            int row = it.second.firs
            int col = it.second.seco

            // Check if we have reac
            // return the current va
            if (row == n - 1 && col
                return diff;

            for (int i = 0; i < 4; i
            {
                // row - 1, col
```

```
                // row, col + 1
                // row - 1, col
                // row, col - 1
                int newr = row + dr[
                int newc = col + dc[

                // Checking validity
                if (newr >= 0 && new
                {
                    // Effort can be
                    // between the h
                    int newEffort =

                    // If the calcul
                    // we update as
                    if (newEffort <
                    {
                        dist[newr][n
                        pq.push({new
                    }
                }
            }
        }
        return 0; // if unreachable
    }
};

int main()
{
    // Driver Code.

    vector<vector<int>> heights = {{

    Solution obj;

    int ans = obj.MinimumEffort(heig

    cout << ans;
    cout << endl;

    return 0;
}
```

Output :

2

Time Complexity: O(4*N*M * log(N*M)) { N*M are the total cells, for each of which we also check 4 adjacent nodes for the minimum effort and additional log(N*M) for insertion-deletion operations in a priority queue }

Where, N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Space Complexity: O(N*M) { Distance matrix containing N*M cells + priority queue in the worst case containing all the nodes (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 distance;
    int row;
    int col;
    public Tuple(int distance,int ro
        this.row = row;
        this.distance = distance;
        this.col = col;
    }
}
class Solution {

    int MinimumEffort(int heights[][

        // Create a priority queue c
        // and their respective dist
        // form {diff, {row of cell,
        PriorityQueue<Tuple> pq =
        new PriorityQueue<Tuple>((x,
```

```
int n = heights.length;
int m = heights[0].length;

// Create a distance matrix 
// unvisited and the dist fo
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[0][0] = 0;
pq.add(new Tuple(0, 0, 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 matri
// and pushing whenever a sh
while(pq.size() != 0) {
    Tuple it = pq.peek();
    pq.remove();
    int diff = it.distance;
    int row = it.row;
    int col = it.col;

    // Check if we have reac
    // return the current va
    if(row == n-1 && col == 
    // row - 1, col
    // row, col + 1
    // row - 1, col
    // row, col - 1
    for(int i = 0;i<4;i++) {
        int newr = row + dr[
        int newc = col + dc[

        // Checking validity
        if(newr>=0 && newc >
```

```
                        // Effort can be
                        // between the h
                        int newEffort =
                        Math.max(
                            Math.abs(hei

                        // If the calcul
                        // we update as
                        if(newEffort < d
                            dist[newr][n
                            pq.add(new T
                        }
                    }
                }
            }
            // If the destination is unr
            return 0;
        }
    }

    class tuf {

        public static void main(String[]

            int[][] heights={{1, 2, 2},

            Solution obj = new Solution(
            int ans = obj.MinimumEffort(

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

Output :

2

Time Complexity: O(4*N*M * log(N*M)) { N*M are the total cells, for each of which we also check 4 adjacent nodes for the minimum effort and additional log(N*M) for insertion-deletion operations in a priority queue }

Where, N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Space Complexity: O(N*M) { Distance matrix containing N*M cells + priority queue in the worst case containing all the nodes (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
G-36: Shortest Distance in a Binary Maze

Next Post »
G-38: Cheapest Flights Within K Stops

Load Comments