48、最小传输时延 II， 考点 or 实现——深度优先搜索 DFS

题目描述

有M*N的节点矩阵，每个节点可以向8个方向（上、下、左、右及四个斜线方向）转发数据包，每个节点转发时会消耗固定时延，连续两个相同时延可以减少一个时延值（即当有K个相同时延的节点连续转发时可以减少K- 1个时延值），

求左上角（0，0）开始转发数据包到右下角（M-1，N- 1）并转发出的最短时延。

输入描述

第一行两个数字，M、N，接下来有M行，每行有N个数据，表示M* N的矩阵。

输出描述

最短时延值。

用例

输入	3 3 0 2 2 1 2 1 2 2 1
输出	3
说明	无

输入	3 3 2 2 2 2 2 2 2 2 2
输出	4
说明	(2 + 2 + 2 - (3-1))

题目解析

本题可以使用深度优先搜索解题。具体逻辑请看代码注释。

JavaScript算法源码

```javascript
/* JavaScript Node ACM模式 控制台输入获取 */
const readline = require("readline");

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

const lines = [];
let m, n;
let matrix;

// 八个方向的偏移量
const offsets = [
  [-1, 0],
  [1, 0],
  [0, -1],
  [0, 1],
  [-1, -1],
  [-1, 1],
  [1, -1],
  [1, 1],
];

rl.on("line", (line) => {
  lines.push(line);

  if (lines.length === 1) {
    [m, n] = lines[0].split(" ").map(Number);
  }

  if (n && lines.length === n + 1) {
    matrix = lines.slice(1).map((line) => line.split(" ").map(Number));
    console.log(getResult());
    lines.length = 0;
  }
});

function getResult() {
  const res = [];
  const path = new Set();
  path.add(0); // 初始时，[0,0]位置已被扫描，因此要将它的一维坐标形式，即 0 * m + 0，加入已扫描过的节点列表path中，避免重复扫
  dfs(0, 0, 0, Infinity, path, res);

  return Math.min.apply(null, res);
}

/**
 * @param {*} matrix 矩阵
 * @param {*} i 当前正在被dfs的节点的横坐标
 * @param {*} j 当前正在被dfs的节点的纵坐标
 * @param {*} delay 已累计的时延值
```

```
 * @param {*} last 上一个节点的时延值，用于和当前节点时延值对比，若相同，则新增时延-1
 * @param {*} path 记录扫描过的节点的位置，避免重复扫描
 * @param {*} res 记录各种从起点到终点的路径的时延值
 */
function dfs(i, j, delay, last, path, res) {
  // 当前节点的时延值
  const cur = matrix[i][j];

  // flag用于标记，当前节点和上一个节点的时延值是否相同，若相同，则新增的时延值要-1
  let flag = cur === last;

  // 如果搜索到了最后一个点，那么就将该路径的时延计算出来，加入到res中，结束分支递归
  if (i === m - 1 && j === n - 1) {
    delay += cur - (flag ? 1 : 0);
    return res.push(delay);
  }

  // 深度优先搜索当前点的八个方向
  for (let offset of offsets) {
    const [offsetX, offsetY] = offset;
    const newI = i + offsetX;
    const newJ = j + offsetY;

    // 将二维坐标，转成一维坐标pos
    const pos = newI * m + newJ;

    // 如果新位置越界，或者新位置已经扫描过，则停止递归
    if (newI >= 0 && newI < m && newJ >= 0 && newJ < n && !path.has(pos)) {
      path.add(pos);
      dfs(
        newI,
        newJ,
        delay + cur - (flag ? 1 : 0), // 当前节点和上一个节点的时延值是否相同，若相同，则新增的时延值要-1
        cur,
        path,
        res
      );
      path.delete(pos);
    }
  }
}
```

Java算法源码

```java
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Scanner;

public class Main {
  static int[][] matrix;
  static int m;
  static int n;

  static int[][] offsets = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}, {-1, -1}, {-1, 1}, {1, -1}, {1, 1}};

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    m = sc.nextInt();
    n = sc.nextInt();

    matrix = new int[m][n];
    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
        matrix[i][j] = sc.nextInt();
      }
    }

    System.out.println(getResult(matrix));
  }

  public static int getResult(int[][] matrix) {
    ArrayList<Integer> res = new ArrayList<>();
    HashSet<Integer> path = new HashSet<>();
    path.add(0);

    dfs(0, 0, 0, Integer.MAX_VALUE, path, res);

    return res.stream().min((a, b) -> a - b).get();
  }

  /**
   * @param i 当前正在被dfs的节点的横坐标
   * @param j 当前正在被dfs的节点的纵坐标
   * @param delay 已累计的时延值
   * @param last 上一个节点的时延值，用于和当前节点时延值对比，若相同，则新增时延-1
   * @param path 记录扫描过的节点的位置，避免重复扫描
   * @param res 记录各种从起点到终点的路径的时延值
   */
  public static void dfs(
      int i, int j, int delay, int last, HashSet<Integer> path, ArrayList<Integer> res) {
    // 当前节点的时延值
    int cur = matrix[i][j];

```

```
    // flag用于标记，当前节点和上一个节点的时延值是否相同，若相同，则新增的时延值要-1
    boolean flag = cur == last;

    // 如果搜索到了最后一个点，那么就将该路径的时延计算出来，加入到res中，结束分支递归
    if (i == m - 1 && j == n - 1) {
      delay += cur - (flag ? 1 : 0);
      res.add(delay);
      return;
    }

    // 深度优先搜索当前点的八个方向
    for (int[] offset : offsets) {
      int newI = i + offset[0];
      int newJ = j + offset[1];

      // 将二维坐标，转成一维坐标pos
      int pos = newI * m + newJ;

      // 如果新位置越界，或者新位置已经扫描过，则停止递归
      if (newI >= 0 && newI < m && newJ >= 0 && newJ < n && !path.contains(pos)) {
        path.add(pos);
        dfs(
            newI,
            newJ,
            delay + cur - (flag ? 1 : 0), // 当前节点和上一个节点的时延值是否相同，若相同，则新增的时延值要-1
            cur,
            path,
            res);
        path.remove(pos);
      }
    }
  }
}
```

Python算法源码

```
import sys

# 输入获取
m, n = map(int, input().split())
matrix = [list(map(int, input().split())) for i in range(m)]

# 八个方向的偏移量
offsets = ((-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1))


def dfs(i, j, delay, last, path, res):
    """
    :param i: 当前正在被dfs的节点的横坐标
    :param j: 当前正在被dfs的节点的纵坐标
    :param delay: 已累计的时延值
    :param last: 上一个节点的时延值，用于和当前节点时延值对比，若相同，则新增时延-1
    :param path: 记录扫描过的节点的位置，避免重复扫描
    :param res: 记录各种从起点到终点的路径的时延值
    """
    # 当前节点的时延值
    cur = matrix[i][j]

    # flag用于标记，当前节点和上一个节点的时延值是否相同，若相同，则新增的时延值要-1
    flag = (cur == last)

    # 如果搜索到了最后一个点，那么就将该路径的时延计算出来，加入到res中，结束分支递归
```

```
    # 如果搜索到了最后一个点，那么就将该路径的时延计算出来，加入到res中，结束分支递归
    if i == m - 1 and j == n - 1:
        delay += cur - (1 if flag else 0)
        return res.append(delay)

    # 深度优先搜索当前点的八个方向
    for offsetX, offsetY in offsets:
        newI = i + offsetX
        newJ = j + offsetY

        pos = newI * m + newJ

        if 0 <= newI < m and 0 <= newJ < n and pos not in path:
            path.add(pos)
            # 当前节点和上一个节点的时延值是否相同，若相同，则新增的时延值要-1
            dfs(newI, newJ, delay + cur - (1 if flag else 0), cur, path, res)
            path.remove(pos)


# 算法入口
def getResult():
    res = []
    path = set()

    # 初始时，[0,0]位置已被扫描，因此要将它的一维坐标形式，即 0 * m + 0，加入已扫描过的节点列表path中，避免重复扫描
    path.add(0)

    dfs(0, 0, 0, sys.maxsize, path, res)
```

```
    dfs(0, 0, 0, sys.maxsize, path, res)

    print(min(res))


# 算法调用
getResult()
```