

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

```
1  /* JavaScript Node ACM模式 控制台输入获取 */
2  const readline = require("readline");
3
4  const rl = readline.createInterface({
5    input: process.stdin,
6    output: process.stdout,
7  });
8
9  const lines = [];
10 let m, n;
11 let matrix;
12
13 // 八个方向的偏移量
14 const offsets = [
15   [-1, 0],
16   [1, 0],
17   [0, -1],
18   [0, 1],
19   [-1, -1],
20   [-1, 1],
21   [1, -1],
22   [1, 1],
23 ];
24
25 rl.on("line", (line) => {
26   lines.push(line);
27
28   if (lines.length === 1) {
29     [m, n] = lines[0].split(" ").map(Number);
30   }
31
32   if (n && lines.length === n + 1) {
33     matrix = lines.slice(1).map((line) => line.split(" ").map(Number));
34     console.log(getResult());
35     lines.length = 0;
36   }
37 });
38
39 function getResult() {
40   const res = [];
41   const path = new Set();
42   path.add(0); // 初始时, [0,0]位置已被扫描, 因此要把它的一维坐标形式, 即  $0 * m + 0$ , 加入已扫描过的节点列表path中, 避免重复扫描
43   dfs(0, 0, 0, Infinity, path, res);
44
45   return Math.min.apply(null, res);
46 }
47
48 /**
49  * @param {*} matrix 矩阵
50  * @param {*} i 当前正在被dfs的节点的横坐标
51  * @param {*} j 当前正在被dfs的节点的纵坐标
52  * @param {*} delay 已累计的时延值
```

```

53  * @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
54  * @param {*} path 记录扫描过的节点的位置，避免重复扫描
55  * @param {*} res 记录各种从起点到终点的路径的时延值
56  */
57  function dfs(i, j, delay, last, path, res) {
58    // 当前节点的时延值
59    const cur = matrix[i][j];
60
61    // 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
62    let flag = cur === last;
63
64    // 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
65    if (i === m - 1 && j === n - 1) {
66      delay += cur - (flag ? 1 : 0);
67      return res.push(delay);
68    }
69
70    // 深度优先搜索当前点的八个方向
71    for (let offset of offsets) {
72      const [offsetX, offsetY] = offset;
73      const newI = i + offsetX;
74      const newJ = j + offsetY;
75
76      // 将二维坐标，转成一维坐标pos
77      const pos = newI * m + newJ;
78
79      // 如果新位置越界，或者新位置已经扫描过，则停止递归
80      if (newI >= 0 && newI < m && newJ >= 0 && newJ < n && !path.has(pos)) {
81        path.add(pos);
82        dfs(
83          newI,
84          newJ,
85          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
86          cur,
87          path,
88          res
89        );
90        path.delete(pos);
91      }
92    }
93  }

```

Java算法源码

```
1 import java.util.ArrayList;
2 import java.util.HashSet;
3 import java.util.Scanner;
4
5 public class Main {
6     static int[][] matrix;
7     static int m;
8     static int n;
9
10    static int[][] offsets = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}, {-1, -1}, {-1, 1}, {1, -1}, {1, 1}};
11
12    public static void main(String[] args) {
13        Scanner sc = new Scanner(System.in);
14
15        m = sc.nextInt();
16        n = sc.nextInt();
17
18        matrix = new int[m][n];
19        for (int i = 0; i < m; i++) {
20            for (int j = 0; j < n; j++) {
21                matrix[i][j] = sc.nextInt();
22            }
23        }
```

```
24
25        System.out.println(getResult(matrix));
26    }
27
28    public static int getResult(int[][] matrix) {
29        ArrayList<Integer> res = new ArrayList<>();
30        HashSet<Integer> path = new HashSet<>();
31        path.add(0);
32
33        dfs(0, 0, 0, Integer.MAX_VALUE, path, res);
34
35        return res.stream().min((a, b) -> a - b).get();
36    }
37
38    /**
39     * @param i 当前正在被dfs的节点的横坐标
40     * @param j 当前正在被dfs的节点的纵坐标
41     * @param delay 已累计的时延值
42     * @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
43     * @param path 记录扫描过的节点的位置, 避免重复扫描
44     * @param res 记录各种从起点到终点的路径的时延值
45     */
46    public static void dfs(
47        int i, int j, int delay, int last, HashSet<Integer> path, ArrayList<Integer> res) {
48        // 当前节点的时延值
49        int cur = matrix[i][j];
```

```

51 // 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
52 boolean flag = cur == last;
53
54 // 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
55 if (i == m - 1 && j == n - 1) {
56     delay += cur - (flag ? 1 : 0);
57     res.add(delay);
58     return;
59 }
60
61 // 深度优先搜索当前点的八个方向
62 for (int[] offset : offsets) {
63     int newI = i + offset[0];
64     int newJ = j + offset[1];
65
66     // 将二维坐标，转成一维坐标pos
67     int pos = newI * m + newJ;
68
69     // 如果新位置越界，或者新位置已经扫描过，则停止递归
70     if (newI >= 0 && newI < m && newJ >= 0 && newJ < n && !path.contains(pos)) {
71         path.add(pos);
72         dfs(
73             newI,
74             newJ,
75             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
76             cur,
77             path,
78             res);
79         path.remove(pos);
80     }
81 }
82 }
83 }

```

Python算法源码

```

1 import sys
2
3 # 输入获取
4 m, n = map(int, input().split())
5 matrix = [list(map(int, input().split())) for i in range(m)]
6
7 # 八个方向的偏移量
8 offsets = ((-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1))
9
10
11 def dfs(i, j, delay, last, path, res):
12     """
13     :param i: 当前正在被dfs的节点的横坐标
14     :param j: 当前正在被dfs的节点的纵坐标
15     :param delay: 已累计的时延值
16     :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
17     :param path: 记录扫描过的节点的位置，避免重复扫描
18     :param res: 记录各种从起点到终点的路径的时延值
19     """
20     # 当前节点的时延值
21     cur = matrix[i][j]
22
23     # 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
24     flag = (cur == last)
25
26     # 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

```

```

26     # 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
27     if i == m - 1 and j == n - 1:
28         delay += cur - (1 if flag else 0)
29         return res.append(delay)
30
31     # 深度优先搜索当前点的八个方向
32     for offsetX, offsetY in offsets:
33         newI = i + offsetX
34         newJ = j + offsetY
35
36         pos = newI * m + newJ
37
38         if 0 <= newI < m and 0 <= newJ < n and pos not in path:
39             path.add(pos)
40             # 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
41             dfs(newI, newJ, delay + cur - (1 if flag else 0), cur, path, res)
42             path.remove(pos)
43
44
45     # 算法入口
46     def getResult():
47         res = []
48         path = set()
49
50         # 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
51         path.add(0)
52
53         dfs(0, 0, 0, sys.maxsize, path, res)
54
55         print(min(res))
56
57
58     # 算法调用
59     getResult()

```