【华为OD机考 统一考试机试C卷】寻找最优的路测线路（C++ Java JavaScript Python）

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

2023年11月份，华为官方已经将 华为OD机考：OD统一考试（A卷 / B卷）切换到 OD统一考试（C卷）和 OD统一考试（D卷） 。根据考友反馈：目前抽到的试卷为B卷或C卷/D卷，其中C卷居多 ，按照之前的经验C卷D卷部分考题会复用A卷/B卷题，博主正积极从考过的同学收集C卷和D卷真题，可以查看下面的真题目录。

真题目录：华为OD机考机试 真题目录（C卷 + D卷 + B卷 + A卷） + 考点说明
专栏：2023华为OD机试(B卷+C卷+D卷)（C++JavaJSPy）
华为OD面试真题精选：华为OD面试真题精选
在线OJ：点击立即刷题，模拟真实机考环境 华为OD机考B卷C卷华为OD机考华为OD机考B卷华为OD机试B卷华为OD机试C卷华为OD机考C卷华为OD机考D卷题目华为OD机考C卷/D卷答案华为OD机考C卷/D卷解析华为OD机考C卷和D卷真题华为OD机考C卷和D卷题解

题目描述

评估一个网络的信号质量，其中一个做法是将网络划分为栅格，然后对每个栅格的信号质量计算。

路测的时候，希望选择一条信号最好的路线（彼此相连的栅格集合）进行演示。

现给出 R 行 C 列的整数数组 Cov，每个单元格的数值 S 即为该栅格的信号质量（已归一化，无单位，值越大信号越好）。

要求从 [0, 0] 到 [R-1, C-1]设计一条最优路测路线。返回该路线得分。

规则：

- 路测路线可以上下左右四个方向，不能对角
- 路线的评分是以路线上信号最差的栅格为准的，例如路径 8→4→5→9 的值为4，该线路评分为4。线路最优表示该条线路的评分最高。

输入描述

一行表示栅格的行数 R

第二行表示栅格的列数 C

第三行开始，每一行表示栅格地图一行的信号值，如5 4 5

输出描述

最优路线的得分

备注

- 1 ≤ R，C ≤ 20
- 0 ≤ S ≤ 65535

用例1

输入

```
3
3
5 4 5
1 2 6
7 4 6
```

输出

```
4
```

说明

> 路线为：5→4→5→6→6

用例2

输入

```
6
5
3 4 6 3 4
0 2 1 1 7
8 8 3 2 7
3 2 4 9 8

```

```
8
4 1 2 0 0
4 6 5 4 3
```

输出

```
1
3
```

说明

> 路线为：3→4→6→3→4→7→7→8→9→4→3→8→8→3→4→4→6→5→4→3

解题思路

使用 广度优先搜索（BFS）+ 二分查找 。

1. **广度优先搜索 (BFS):**
   - `bfs` 函数实现了广度优先搜索算法。它的目的是检查是否存在一条从网格的左上角（起点）到右下角（终点）的路径，且路径上每个单元的信号强度都至少为 `minSignal` 。
   - 它首先检查起点和终点的信号强度，如果任何一个小于 `minSignal` ，则返回false。
   - 使用一个队列来存储待访问的单元格，从起点开始搜索。
   - 对于队列中的每个元素，它会检查从该点出发可以到达的四个方向（上下左右）。如果相邻单元的信号强度满足要求且未被访问过，则将其添加到队列中。
   - 这个过程会一直进行，直到找到一条到达终点的有效路径，或者队列变空（没有路径满足条件）。
2. **二分查找:**
   - `binarySearch` 函数使用二分查找来确定满足条件的最大 `minSignal` 值。
   - 它在最小可能信号强度（ `minSignal` ）和最大可能信号强度（ `maxSignal` ）之间进行搜索。
   - 在每次迭代中，它会计算当前范围的中间值 `mid` ，然后使用BFS检查是否存在一条满足 `mid` 作为最小信号强度的路径。
   - 如果存在这样的路径，它会尝试更高的信号强度；如果不存在，则降低信号强度。
   - 通过不断调整搜索范围，二分查找最终确定了可以找到有效路径的最大 `minSignal` 值。

C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <climits>

using namespace std;

// 定义一个类表示网格中的一个单元格
class Cell {
public:
    int row, col;
    Cell(int r, int c) : row(r), col(c) {} // 构造函数，初始化行和列
};

// 使用广度优先搜索（BFS）检查是否存在一条从起点到终点的路径，路径上所有单元格的信号质量都不低于minSignal
bool bfs(const vector<vector<int>>& Cov, int minSignal) {
    int R = Cov.size(), C = Cov[0].size();
    // 如果起点或终点的信号质量低于minSignal，直接返回false
    if (Cov[0][0] < minSignal || Cov[R - 1][C - 1] < minSignal) {
        return false;
    }

    // visited数组用于记录哪些单元格已经被访问过，避免重复访问
    vector<vector<bool>> visited(R, vector<bool>(C, false));
    queue<Cell> queue;
    queue.push(Cell(0, 0));
    visited[0][0] = true;

    // dr和dc数组用于表示从当前单元格向四个方向（上下左右）移动的行和列的变化量
    int dr[4] = {1, -1, 0, 0};
    int dc[4] = {0, 0, 1, -1};

    while (!queue.empty()) {
        Cell cell = queue.front();
        queue.pop();
        // 如果到达终点，返回true
        if (cell.row == R - 1 && cell.col == C - 1) {
            return true;
        }

```

```
        // 否则，尝试向四个方向移动
        for (int i = 0; i < 4; i++) {
            int nr = cell.row + dr[i];
            int nc = cell.col + dc[i];

            // 如果新的单元格在网格内，且没有被访问过，且信号质量不低于minSignal，将其加入队列并标记为已访问
            if (nr >= 0 && nr < R && nc >= 0 && nc < C && !visited[nr][nc] && Cov[nr][nc] >= minSignal) {
                queue.push(Cell(nr, nc));
                visited[nr][nc] = true;
            }
        }
    }

    // 如果没有找到有效路径，返回false
    return false;
}

// 使用二分搜索找到最大的满足条件的信号质量
int binarySearch(const vector<vector<int>>& Cov, int low, int high) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        // 如果存在一条有效路径，尝试更高的信号质量
        if (bfs(Cov, mid)) {
            low = mid + 1;
        } else { // 否则，降低信号质量
            high = mid - 1;
        }
    }
    // 返回最大的满足条件的信号质量
    return high;
}

int main() {
    int R, C;
    cin >> R >> C;
    vector<vector<int>> Cov(R, vector<int>(C));

    int minSignal = INT_MAX;
    int maxSignal = INT_MIN;
```

```
    // 读取网格数据，并记录信号质量的最小值和最大值
    for (int i = 0; i < R; i++) {
        for (int j = 0; j < C; j++) {
            cin >> Cov[i][j];
            minSignal = min(minSignal, Cov[i][j]);
            maxSignal = max(maxSignal, Cov[i][j]);
        }
    }

    // 输出最大的满足条件的信号质量
    cout << binarySearch(Cov, minSignal, maxSignal) << endl;
    return 0;
}
```

Java

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class Main {

    // 定义一个内部类表示网格中的一个单元格
    static class Cell {
        int row, col;

        Cell(int row, int col) {
            this.row = row;
            this.col = col;
        }
    }

    // 使用广度优先搜索（BFS）检查是否存在一条从起点到终点的路径，路径上所有单元格的信号质量都不低于minSignal
    private static boolean bfs(int[][] Cov, int minSignal) {
        int R = Cov.length, C = Cov[0].length;
        // 如果起点或终点的信号质量低于minSignal，直接返回false
        if (Cov[0][0] < minSignal || Cov[R - 1][C - 1] < minSignal) {
            return false;
        }

```

```
        // visited数组用于记录哪些单元格已经被访问过，避免重复访问
        boolean[][] visited = new boolean[R][C];
        Queue<Cell> queue = new LinkedList<>();
        queue.add(new Cell(0, 0));
        visited[0][0] = true;

        // dr和dc数组用于表示从当前单元格向四个方向（上下左右）移动的行和列的变化量
        int[] dr = {1, -1, 0, 0};
        int[] dc = {0, 0, 1, -1};

        while (!queue.isEmpty()) {
            Cell cell = queue.poll();
            // 如果到达终点，返回true
            if (cell.row == R - 1 && cell.col == C - 1) {
                return true;
            }

            // 否则，尝试向四个方向移动
            for (int i = 0; i < 4; i++) {
                int nr = cell.row + dr[i];
                int nc = cell.col + dc[i];

                // 如果新的单元格在网格内，且没有被访问过，且信号质量不低于minSignal，将其加入队列并标记为已访问
                if (nr >= 0 && nr < R && nc >= 0 && nc < C && !visited[nr][nc] && Cov[nr][nc] >= minSignal) {
                    queue.add(new Cell(nr, nc));
                    visited[nr][nc] = true;
                }
            }
        }

        // 如果没有找到有效路径，返回false
        return false;
    }

    // 使用二分搜索找到最大的满足条件的信号质量
    private static int binarySearch(int[][] Cov, int low, int high) {
        while (low <= high) {
            int mid = low + (high - low) / 2;
            // 如果存在一条有效路径，尝试更高的信号质量
            if (bfs(Cov, mid)) {
```

```
                low = mid + 1;
            } else { // 否则，降低信号质量
                high = mid - 1;
            }
        }
        // 返回最大的满足条件的信号质量
        return high;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int R = scanner.nextInt();
        int C = scanner.nextInt();
        int[][] Cov = new int[R][C];

        int minSignal = Integer.MAX_VALUE;
        int maxSignal = Integer.MIN_VALUE;

        // 读取网格数据，并记录信号质量的最小值和最大值
        for (int i = 0; i < R; i++) {
            for (int j = 0; j < C; j++) {
                Cov[i][j] = scanner.nextInt();
                minSignal = Math.min(minSignal, Cov[i][j]);
                maxSignal = Math.max(maxSignal, Cov[i][j]);
            }
        }
        scanner.close();

        // 输出最大的满足条件的信号质量
        System.out.println(binarySearch(Cov, minSignal, maxSignal));
    }
}
```

javaScript

```
// 导入所需的库
const readline = require('readline');

// 创建readline.Interface实例
const rl = readline.createInterface({
```

```
    input: process.stdin,
    output: process.stdout
});

// 存储所有输入行的数组
let lines = [];
rl.on('line', (line) => {
    lines.push(line);
});

// 在输入完毕后执行主要逻辑
rl.on('close', () => {

    let R = parseInt(lines[0]);
    let C = parseInt(lines[1]);
    let Cov = [];
    for (let i = 0; i < R; i++) {
        Cov.push(lines[i + 2].split(' ').map(Number));
    }

    let minSignal = Math.min(...Cov.map(row => Math.min(...row)));
    let maxSignal = Math.max(...Cov.map(row => Math.max(...row)));

    // 输出最大的满足条件的信号质量
    console.log(binary_search(Cov, minSignal, maxSignal));
});

// 定义一个类表示网格中的一个单元格
class Cell {
    constructor(row, col) {
        this.row = row;
        this.col = col;
    }
}

// 使用广度优先搜索（BFS）检查是否存在一条从起点到终点的路径
// 路径上所有单元格的信号质量都不低于minSignal
function bfs(Cov, minSignal) {
    let R = Cov.length, C = Cov[0].length;
    // 如果起点或终点的信号质量低于minSignal，直接返回false
```

```
    if (Cov[0][0] < minSignal || Cov[R - 1][C - 1] < minSignal) {
        return false;
    }

    // visited数组用于记录哪些单元格已经被访问过，避免重复访问
    let visited = Array.from({ length: R }, () => Array(C).fill(false));
    let queue = [];
    queue.push(new Cell(0, 0));
    visited[0][0] = true;

    // dr和dc数组用于表示从当前单元格向四个方向（上下左右）移动的行和列的变化量
    let dr = [1, -1, 0, 0];
    let dc = [0, 0, 1, -1];

    while (queue.length > 0) {
        let cell = queue.shift();
        // 如果到达终点，返回True
        if (cell.row == R - 1 && cell.col == C - 1) {
            return true;
        }

        // 否则，尝试向四个方向移动
        for (let i = 0; i < 4; i++) {
            let nr = cell.row + dr[i];
            let nc = cell.col + dc[i];

            // 如果新的单元格在网格内，且没有被访问过，且信号质量不低于minSignal，将其加入队列并标记为已访问
            if (nr >= 0 && nr < R && nc >= 0 && nc < C && !visited[nr][nc] && Cov[nr][nc] >= minSignal) {
                queue.push(new Cell(nr, nc));
                visited[nr][nc] = true;
            }
        }
    }

    // 如果没有找到有效路径，返回False
    return false;
}

// 使用二分搜索找到最大的满足条件的信号质量
function binary_search(Cov, low, high) {

```

```
    while (low <= high) {
        let mid = Math.floor(low + (high - low) / 2);
        // 如果存在一条有效路径，尝试更高的信号质量
        if (bfs(Cov, mid)) {
            low = mid + 1;
        } else {  // 否则，降低信号质量
            high = mid - 1;
        }
    }
    // 返回最大的满足条件的信号质量
    return high;
}
```

Python

```
from collections import deque

# 定义一个类表示网格中的一个单元格
class Cell:
    def __init__(self, row, col):
        self.row = row
        self.col = col

# 使用广度优先搜索（BFS）检查是否存在一条从起点到终点的路径
# 路径上所有单元格的信号质量都不低于minSignal
def bfs(Cov, minSignal):
    R, C = len(Cov), len(Cov[0])
    # 如果起点或终点的信号质量低于minSignal，直接返回false
    if Cov[0][0] < minSignal or Cov[R - 1][C - 1] < minSignal:
        return False

    # visited数组用于记录哪些单元格已经被访问过，避免重复访问
    visited = [[False for _ in range(C)] for _ in range(R)]
    queue = deque()
    queue.append(Cell(0, 0))
```

```python
    visited[0][0] = True

    # dr和dc数组用于表示从当前单元格向四个方向（上下左右）移动的行和列的变化量
    dr = [1, -1, 0, 0]
    dc = [0, 0, 1, -1]

    while queue:
        cell = queue.popleft()
        # 如果到达终点，返回True
        if cell.row == R - 1 and cell.col == C - 1:
            return True

        # 否则，尝试向四个方向移动
        for i in range(4):
            nr, nc = cell.row + dr[i], cell.col + dc[i]

            # 如果新的单元格在网格内，且没有被访问过，且信号质量不低于minSignal，将其加入队列并标记为已访问
            if 0 <= nr < R and 0 <= nc < C and not visited[nr][nc] and Cov[nr][nc] >= minSignal:
                queue.append(Cell(nr, nc))
                visited[nr][nc] = True

    # 如果没有找到有效路径，返回False
    return False

# 使用二分搜索找到最大的满足条件的信号质量
def binary_search(Cov, low, high):
    while low <= high:
        mid = low + (high - low) // 2
        # 如果存在一条有效路径，尝试更高的信号质量
        if bfs(Cov, mid):
            low = mid + 1
        else:  # 否则，降低信号质量
            high = mid - 1
    # 返回最大的满足条件的信号质量
    return high

# 主函数
def main():
    R = int(input())
    C = int(input())
```

```
    Cov = [list(map(int, input().split())) for _ in range(R)]

    minSignal = min(min(row) for row in Cov)
    maxSignal = max(max(row) for row in Cov)

    # 输出最大的满足条件的信号质量
    print(binary_search(Cov, minSignal, maxSignal))

if __name__ == "__main__":
    main()
```

文章目录


机考真题
华为OD
No No No
CSDN @算法大师