信息学奥赛笔记03

广度优先搜索(Breadth-First Search)和相关练习

广度优先搜索

概念

广度优先搜索(BFS)也叫做宽度优先搜索，从宏观上看是一种图遍历的算法。这也是很多重要图的算法的前置知识，如Dijkstra单源最短路及Prim最小生成树算法。和深度优先搜索(DFS)相似的是，BFS也是一种盲目搜寻法，目的是为了能够系统的，不重不漏地，快速地，遍历图中的所有的结点，以找寻结果。为什么叫做盲目呢？因为它并不考虑结果可能的位置，彻底的搜索整张图，直到找到答案为止。

策略

BFS所采用的策略通俗易懂：**越早被访问到的结点，它的邻居结点越早被访问到。**虽然一句话就能够概括广度优先搜索的原理，但是想要彻底的对这句话吃透，还是需要下一些功夫的。大体上我们可以将BFS的搜寻步骤划分如下：

① 将根结点加入到搜寻的队列，开始执行BFS

② 取出队头结点`node`作为当前访问到的结点，将该结点标记为“访问”，判断`node`是否为目标结点，如果**是**则执行步骤⑤，**否则**执行步骤③

③ 寻找`node`的所有邻居结点，如果邻居结点已经被访问过则跳过，否则则将其加入队列

④ 重复执行步骤②至步骤④，直到寻找到目标点时执行步骤⑤

⑤ 输出结果

细节

为什么广度优先搜索需要使用**队列**这个元素呢？因为队列的实现策略，刚好符合BFS的策略，也就是**先进先出(First In First Out)**，先被访问到的结点，它的邻居结点先被加入队列，也会先出队列被访问到，于是能够先访问到该结点的邻居结点。这句话可能比较绕口，大家只需要知道，BFS的本质就是**FIFO(先进先出)**，队列的本质也是**FIFO**，所以，只要是涉及到`BFS`的问题，基本上是和**队列**这个数据结构不可分割的，这也是能够体现出，数据结构与算法相辅相成，互相成就的一点。

在步骤①中，如果不将根节点加入队列的话，就没有开始的结点，没有启动推动算法的动力，也就不会发生一连串的连锁反应，所以该在何处加入根节点？加入几个根节点？加入根节点的方式是什么？这是各位同学们解决BFS问题需要最先思考的

在步骤②中，将当前结点标记为访问过，可以使用标记数组，也可以使用map，set实现，具体该用哪个，可以根据题目的需求来选择，但是没有标记数组是一定不可行的，大家可以想一想如果有两个结点相邻，从`A`结点访问到`B`结点，将`B`结点加入队列，但是不将`A`结点标记为`已访问`，此时对于`B`结点来说，它会优先找到它的邻居`A`结点，搜索就在这两个结点之间往复运作，会导致超时，栈溢出，死循环，所以对已访问过的结点做标记，这也是求解BFS问题的重中之重。

在步骤③中，需要找寻当前结点的邻居结点，一般地，从一层到下一层的途径是固定的，比如说四向搜索，八向搜索，国际象棋马搜索，所以为了减少代码量以及使得代码思路更加清晰明了，我们需要使用方向数组 + 判断边界的方式来处理下一结点的情况，这种问题从DFS开始大家就已经经常练习，在这就不多展开说了。

那么，根据一段时间的学习后，同学们会发现，无论是BFS还是DFS，其实都是机械化，公式化的，而这种思想却又是强大，常考的，但同时对代码能力的要求也非常高，所以希望各位同学能够在课后多多练习，找到写搜索题目的感觉。

应用范围

无论是图，还是树，还是矩阵，所有的结点对于根节点来说都有一个不可跳脱的名词——**深度**，也可以理解为这个结点对于根节点的**距离**。而深度优先搜索DFS是更侧重于将一条路径走到底，而BFS更侧重于把当前这一层找到完全，不重不漏，所以两种不同的搜索策略会带来两种不同的结果，DFS搜索的更加彻底和完全，属于地毯式搜索，适合求解：**一共有，数量为，全排列，最坏的情况为**，这种问题；而bfs在找到答案时，一定是最早访问到答案的结点被判断到，也就是说，一旦从队头取出结点，发现该结点就是答案，就意味着这一定是第一次访问到答案，所以更适合求：**最短步数，最少操作次数，最高收益**这种最值问题，所以当你判断出一道题目应该使用搜索的时候，不妨看看这道题目要求的内容是什么，来根据要求的内容选择合适的搜索方法，这才是解题的关键。

伪代码模板

```
void bfs() {
    queue<数据类型> q;
    数据类型 vis[m][n]; // 标记数组
    数据类型 ans[m][n]; // 答案数组
    q.push(起始点);
    while (q.size()) {
        取出队头元素作为当前结点node;
        q.pop();
        vis[队头元素] = true; // 给当前访问到的点做上标记
        if (node == 目标点) {
            //输出答案
            return;
        }
        for (下一个点的方向数量) {
            计算下一个结点的坐标;
            判断该坐标是否越界/可行
            q.push(下一个结点);
            ans[下一节点] = ans[这一节点] + 路径开销;
        }
    }
}
```

课上习题

[U413931] 瘟疫扩散 I https://www.luogu.com.cn/problem/U413931

题目描述

给你一个$n\times m$的矩阵，并输入一个瘟疫的坐标(x,y)，这个瘟疫**每秒**会扩散`上下左右`四格，请你输出一个$n\times m$的矩阵，每一格表示瘟疫最短需要多少秒会扩散到这一格。

输入格式

四个整数，n，m，x，y分别代表矩阵的长宽及瘟疫的坐标。

输出格式

一个$n\times m$的矩阵，代表答案。

样例 #1

样例输入 #1

```
3 3 1 1
```

样例输出 #1

```
2 1 2
1 0 1
2 1 2
```

提示

对于100%的数据，有 $1 \leq n,m \leq 1000$。

思路分析

> 这道题作为bfs的模板题，只需要将瘟疫坐标作为根结点加入队列，按照bfs的模板写全代码即可，因为瘟疫会扩散上下左右四格，所以这是一道四向系列搜索，注意方向数组不要写错。

```cpp
#include <bits/stdc++.h>
using namespace std;
const int dx[] = {1, -1, 0, 0};
const int dy[] = {0, 0, 1, -1}; // 方向数组
int n, m, x, y;
int main() {
    cin >> n >> m >> x >> y;
    queue<pair<int, int> > q;
    vector<vector<int> > a(n, vector<int>(m, -1));
    q.push({x, y}); // 根节点加入
    a[x][y] = 0;    // 答案数组的根节点设置
    while (q.size()) {
        int x = q.front().first, y = q.front().second; q.pop();  // 取出当前结点
        for (int d = 0; d < 4; d++) {
            int mx = x + dx[d], my = y + dy[d]; // 计算下一结点坐标
            if (mx < 0 || mx == n || my < 0 || my == m || a[mx][my] >= 0) continue;  //越界和被访问则跳过
            q.push({mx, my});  // 下一结点入队
            a[mx][my] = a[x][y] + 1;  // 处理答案数组的步数
```

```
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) cout << a[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

瘟疫扩散II

题目描述

给你一个$n\times m$的矩阵，并输入一个瘟疫的坐标(x,y)，这个瘟疫**每秒**会扩散`上下左右`四格，矩阵中一共有K个障碍物阻碍了瘟疫的行径。请你输出一个$n\times m$的矩阵，每一格表示瘟疫最短需要多少秒会扩散到这一格。如果瘟疫无法扩散到这一格，请输出`-1`。

输入格式

第一行五个整数，n，m，k，x，y分别代表矩阵的长宽及瘟疫的坐标。

接下来`k`行，每行`2`个整数x_i，y_i，表示障碍物的坐标。

输出格式

一个$n\times m$的矩阵，代表答案。

样例 #1

样例输入 #1

```
3 3 2 1 1
1 0
0 1
```

样例输出 #1

```
-1 -1 2
-1 0 1
2 1 2
```

提示

对于100%的数据，有$1\leq x<n,y<m\leq 1000$，$0\leq k\leq n\times m$，$0\leq x_i\leq n,0\leq y_i\leq m$。

思路分析

> 和瘟疫扩散 I 最大的区别在于，这道题中新增了一些障碍物，我们在判断进行到下一个格子的时候，只要发现目标格子是障碍物就跳过，所以我们可以把答案数组和统计障碍物的值单独区分开，单独判断即可。

用集合的方法

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, m, k, x, y;
    int dx[] = {1, -1, 0, 0};
    int dy[] = {0, 0, 1, -1};
    cin >> n >> m >> k >> x >> y;
    queue<pair<int, int> > q;
    set<pair<int, int> > s;
    vector<vector<int> > a(n, vector<int>(m, -1));
    q.push({x, y});
    a[x][y] = 0;
    while (k--) {
        cin >> x >> y;
        s.insert({x, y});  //用集合存储障碍物
    }
    while (q.size()) {
        int x = q.front().first, y = q.front().second; q.pop();
        for (int d = 0; d < 4; d++) {
            int mx = x + dx[d], my = y + dy[d];
            if (mx < 0 || mx == n || my < 0 || my == m || a[mx][my] >= 0 || s.count({mx, my})) continue;  //发现是障碍物就跳过
            q.push({mx, my});
            a[mx][my] = a[x][y] + 1;
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) cout << a[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

用标记数组的方法

```cpp
#include<bits/stdc++.h>
using namespace std;
int n, m, k, x, y, u, v;
int a[1005][1005], vis[1005][1005];
const int dx[] = {1, -1, 0, 0}, dy[] = {0, 0, 1, -1};
void bfs() {
    queue<pair<int, int> > q;
    q.push({x, y}); // 根节点入队
    a[x][y] = 0;  // 初始化根节点
    vis[x][y] = 1;  // 将根结点标记为访问
    while (q.size()) {
        int cx = q.front().first, cy = q.front().second; q.pop();
        for (int d = 0; d < 4; d++) {
            int mx = cx + dx[d], my = cy + dy[d];  // 下一个坐标
            if(mx < 0 || mx == n || my < 0 || my == m || vis[mx][my]) continue;  // 已经访问过或者是障碍物，跳过
            q.push({mx, my});
            vis[mx][my] = 1; // 提前为下一个点插上标记，以免别的点再次将目标点加入一次队列导致一个点被加入队列2次
```

```
            a[mx][my] = a[cx][cy] + 1;  // 答案数组更新
        }
    }
}
int main() {
    cin >> n >> m >> k >> x >> y;
    for (int i = 0; i < k; i++) {
        cin >> u >> v;
        vis[u][v] = 1;  //标记有障碍物的格子为已访问
    }
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            a[i][j] = -1;   // 将答案数组全部设为-1
    bfs();
    for (int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) cout << a[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```