数据结构：Broadcast

姓名：鲁国锐　　学号：17020021031　　专业：电子信息科学与技术

2019 年 5 月 16 日

Contents

1 问题分析

1.1 题目描述

某广播公司要在一个地区架设无线广播发射装置。该地区共有 n 个小镇，每个小镇都要安装一台发射机并播放各自的节目。

不过，该公司只获得了 $FM104.2$ 和 $FM98.6$ 两个波段的授权，而使用同一波段的发射机会互相干扰。已知每台发射机的信号覆盖范围是以它为圆心，$20km$ 为半径的圆形区域，因此，如果距离小于 $20km$ 的两个小镇使用同样的波段，那么它们就会由于波段干扰而无法正常收听节目。现在给出这些距离小于 $20km$ 的小镇列表，试判断该公司能否使得整个地区的居民正常听到广播节目。

1.2 输入

第一行为两个整数 n，m，分别为小镇的个数以及接下来小于 $20km$ 的小镇对的数目。接下来的 m 行，每行 2 个整数，表示两个小镇的距离小于 $20km$（编号从 1 开始）。

1.3 输出

如果能够满足要求，输出 1，否则输出 -1。

1.4 问题分析

我们可以分三种情况分析：

1. 距离小于 20 千米的村庄连成一个边数为偶数的多边形且对角线不连通，此时我们会发现只需要令相邻村庄交替使用两个波段就可以避免冲突；
2. 距离小于 20 千米的村庄连成一个边数为奇数的多边形且对角线不连通，此时一定会产生冲突；
3. 距离小于 20 千米的村庄连成一个多边形且对角线连通，此时对角线会将多边形划分成多个更小的对角线不连通的多边形，此时我们把问题化归为了对第一第二种情况的讨论。

综合以上三种情况我们发现，判断是否会产生冲突问题就被转化成了寻找图中是否存在边数为奇数的多边形的问题。

2 解决方案

虽然我们已经分析出了是否会产生冲突的充分必要条件，但是直接用递归来穷举所有最小闭环成本太高。所以1.4节中的结论虽然简明但不实用。

所以这里我们转换思路，采取广度优先搜索的方法。我们先找到一个非孤立的村庄，然后以它为起点进行广度优先搜索。在搜索的过程中，要确保跟当前村庄直接连接的为被发现的村庄使用的波段跟自己不同。这里我们用一个名为 *status* 的 *vector* 来记录个村庄的状态：0 表示为发现；1、-1 分别表示使用的两种波段。

用这种方法判断是否会产生冲突的条件就是对每一个访问到的村庄，遍历其所有邻接村庄，只要找到一个波段跟当前村庄相同的，就说明一定会产生冲突；若成功对这个连通域进行了一次遍历，则说明在该连通域中不会产生冲突。**注意只是在该连通域中不会产生冲突，可能还存在着其它连通域，我们需要对它们一一进行判断。**

3 算法设计

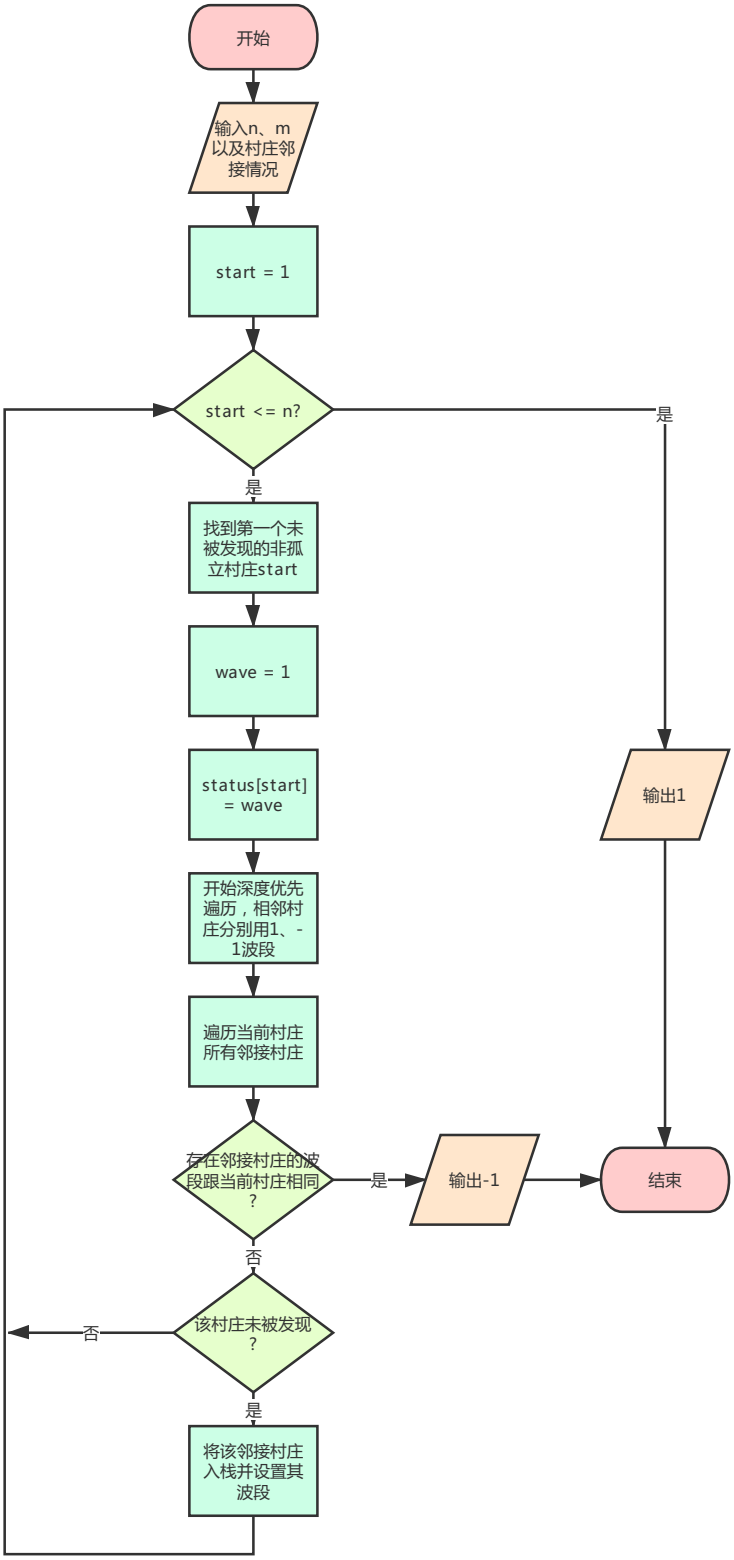

图 1: *main* 函数流程图

4 编程实现[1]

Listing 1: boradcast.cpp

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;


bool linked(vector< vector<int> > &towns, int i)
{
    for (int j = 1; j < towns[i].size(); j++)
    {
        if (towns[i][j])
            return true;
    }
    return false;
}

int main()
{
    int n, m;
    cin >> n >> m;

    // 0 -> undiscovered; 1, -1 -> wave band
    vector<int> status(n+1, 0);
    vector< vector<int> > towns(n+1, vector<int>(n+1, 0));

    for (int i = 0; i < m; i++)
    {
        //cout << towns.size() << "  " << towns[0].size() << endl;
        int t1, t2;
        cin >> t1 >> t2;
        towns[t1][t2] = 1;
        towns[t2][t1] = 1;
    }

    int start = 1;
    while(start <= n)
    {
            //find the first town which is not isolated
            for (; start <= n; start++)
            {
                    if (linked(towns, start) && (status[start] == 0))
            {
                //cout << status[start] << endl;
                break;
            }
        }
```

[1]代码也可从这里得到：https://github.com/chenfeng123456/CourseInOUC/blob/master/algorithm/broadcast/broadcast.cpp

```
47              if (start > n)
48              {
49                      //cout << "start > n" << endl;
50                      cout << 1 << endl;
51                      return 0;
52              }
53              //cout << "start = " << start << endl;
54
55              int wave = 1;
56              status[start] = wave;
57              queue<int> q;
58              q.push(start);
59              while (!q.empty())
60              {
61                      wave = -wave;
62                      int v = q.front();
63                      q.pop();
64                  //cout << endl << "v = " << v << endl;
65
66                      for (int u = 1; u <= n; u++)
67                      {
68                          if (towns[v][u])
69                          {
70                              if (status[u] == status[v])
71                              {
72                                  //cout << "status[" << u << "] = " << status[u] << endl;
73                                  //cout << "status[" << v << "] = " << status[v] << endl;
74                                  cout << -1 << endl;
75                                  return 0;
76                              }
77                              else if (status[u] == 0)
78                              {
79                                  q.push(u);
80                                  status[u] = wave;
81                              }
82                          }
83                      }
84              }
85      }
86      cout << 1 << endl;
87
88      return 0;
89  }
```

5 结果分析

5.1 结果展示

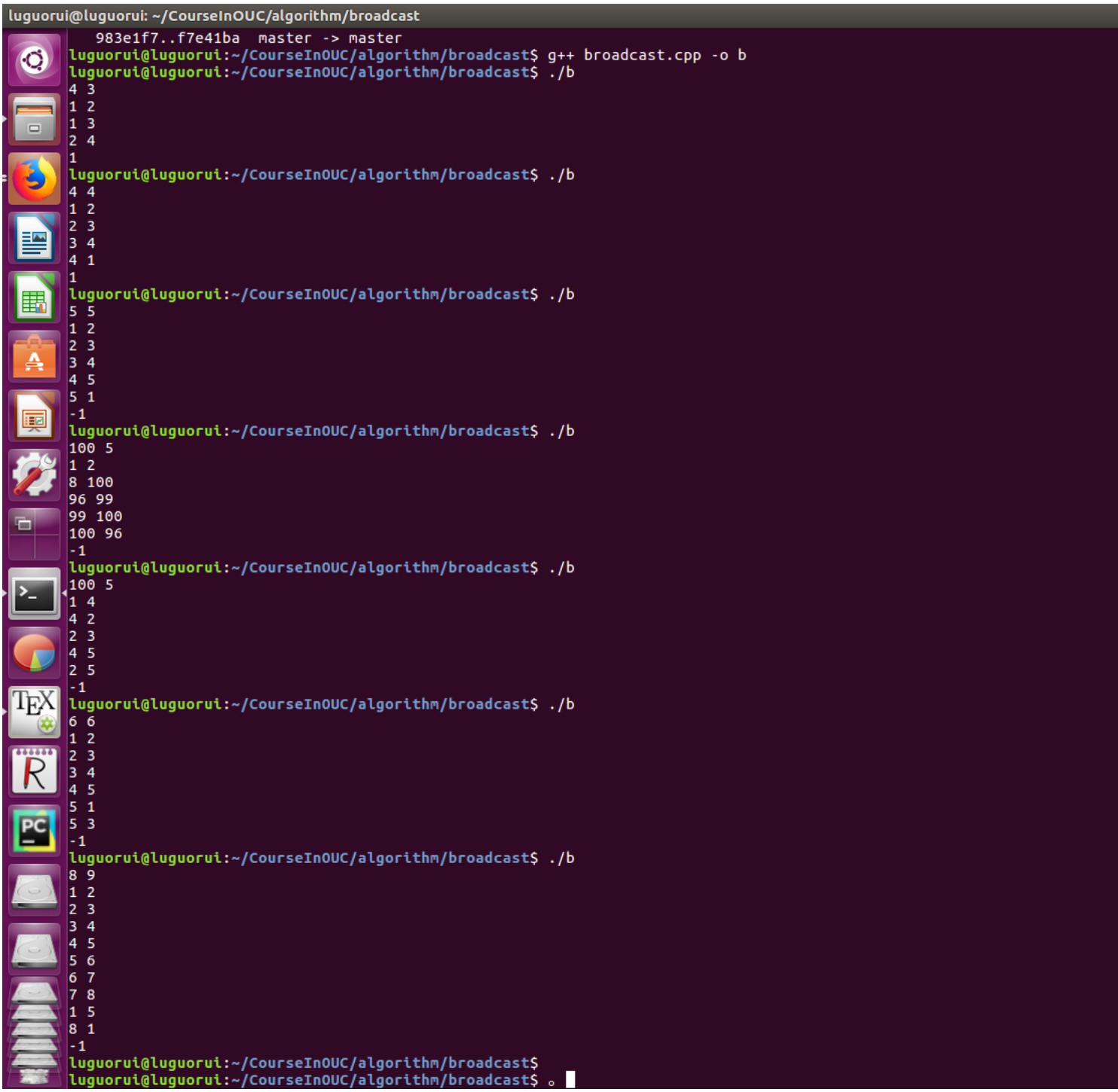

图 2: 结果 1

5.2 可行性分析

这道题相对于前面的作业来说，虽然难度不大，但其算法的可行性却不是一目了然的。之前的作业如祖玛、表达式树，甚至是 *The Art of Compression*，它们尽管有的难度相当大，但是每一步的算法可行性都是显然的，我们在做的时候很清楚这样一定可以解决问题。

然而这题以及之前做的列车调度问题则不同，虽然我们可能很快就想出了算法，但是在实现的过程中我们却始终不确定这样是否一定能得到正确的结果。

以列车调度为例，我们在对栈进行 *pop* 操作时，会先检查一下栈顶元素是否与给定序列中对应元素相同，若不相同则直接退出循环并输出失败信息。然而值得商榷的是，仅仅以栈顶元素与给定序列中相应元素是否相同为条件是否充分？即是否可能存在另外一种 *push*、*pop* 的组合能够得到与给定序列一致的栈混洗？换句话说，就是对解的唯一性问题的讨论。如果只有一种 *push*、*pop* 的组合方式能够得到给定的栈混洗，那么我们的判断条件就是正确的。因为到达目的的路只有一条，而现在如果这唯一的一条路都走不通的话，那就说明肯定无法到达目的地。

我们可以来简单地证明一下。首先 *push*、*pop* 一定是成对出现的，我们可以把它等效为一个括号匹配的问题：*push* 为左括号，*pop* 为右括号。为了方便讨论，这里我们约定 $($ 表示 *push* 了第 i 节车厢，$)^{(i)}$ 表示 *pop* 了第 i 节车厢。

其次我们再来考虑一下栈混洗的顺序到底与什么有关。首先我们可以确定，$(^{(i)}$ 出现的相对顺序是固定的。即它们只能按照 i 从小到大的顺序排列。因为车厢的初始顺序就是这样，它们只能按照这样的相对顺序一个个出栈。因此我们可以断定，栈混洗的顺序与 $(^{(i)}$ 是没有关系的。

既然与 $(^{(i)}$ 没有关系，那就只可能与 $)^{(i)}$ 有关了。事实上，仔细思考以后我们会发现，栈混洗的顺序是与插入在 $(^{(i)}$ 序列中的位置有关的，它插入的位置直接决定了 $)^{(i)}$ 上的 i 等于多少。

举个例子。我们给定一个 $(^{(i)}$ 序列：

$$(^{(1)}(^{(2)}(^{(3)}(^{(4)}(^{(5)} \tag{1}$$

而要得到的栈混洗为：32154。我们先来看一下第一个右括号应该插在哪个地方。

我们已经看到栈混洗的第一个元素为 3，也就是说 $)^{(i)}$ 的上标应该为 3，即第一个 *pop* 所弹出的元素必须为 3。考虑一下，第一个右括号插入的位置只能在 $(^{(3)}$ 的后面：

$$(^{(1)}(^{(2)}(^{(3)})^{(3)}(^{(4)}(^{(5)} \tag{2}$$

顺着刚才的思路，我们知道第二个右括号必须和 $(^{(2)}$ 是匹配的。而要和 $(^{(2)}$ 匹配，此时有两个位置可以插入，即 $(^{(3)})^{(3)}$ 前后两个空。但我们立马就能发现第二个右括号不能插在 $(^{(3)})^{(3)}$ 前面的那个空，因为一旦这样做了，第一个出栈的元素就会变成 2。这也告诉我们，$)^{(i)}$ 的排列的相对顺序必须和栈混洗一致。所以第二个括号也只有唯一的一个位置可供插入。

为了使问题进一步简化，我们可以采取“**化归**”的思想。在第一个右括号插入以后，我们分析出之后的右括号不能插入在 $(^{(3)})^{(3)}$ 前面的那个空（实际上它们前面所有的空都不能插入），那么我们发现，此时已经匹配的括号对于后续括号的匹配除了划定了一个分界线以外没有任何影响，但只要给定的栈混洗是合法的，我们绝不可能跨越这条分界线。**因而我们可以大胆地把已经匹配的括号擦掉**。这是序列就变成了：

$$(^{(1)}(^{(2)}(^{(4)}(^{(5)} \tag{3}$$

再次强调一下，这里 2 和 4 之间的空实际上是 $(^{(3)})^{(3)}$ 后面的那个空，因为前面的那个不能插入。

这时我们发现，问题又回到了**最初的状态**，这正是我们化归所期望的结果：插入第二个右括号的问题变成了同插入第一个括号时一样的情况。所以引用插入第一个括号时的结论，第二个括号也只有唯一的位置插入，并且以此类推，其后所有右括号都只有唯一的一个位置插入。**既然每一步都是唯一确定的，那么不难得出整个问题的解也就是唯一确定的了**。至此，我们就**不那么严谨地**证明了列车调度问题的可行性。

可以看出，简单问题的算法可行性不一定就是一目了然的。回到这一次的作业上，我们也可以提出类似的问题：是否会存在以不同的村庄为起点出发最后能得出不同的结果的情况。这时我们会发现1.4节中所做的讨论不是没有用的，因为当一个村庄发现它的邻接村庄使用了跟自己相同的波段，其本质就在于**这两个所在的某一个最小闭环是一个边数为奇数的多边形**。既然如此，不论我选择从哪里出发，该闭环是一个边数为奇数的多边形的事实不会改变，所以我们得出的结论自然也不会改变。

6 总结体会

尽管知不知道上面的可行性分析，对我们实现代码不会有任何影响，但我还是认为应该多考虑考虑这方面。这样在实现的时候也会更有底气，同时也避免了做到最后发现是“无用功”的情形。